KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Dataflow-inspired parallel and distributed computing

Practical techniques and real-world use cases

Zubair Wadood Bhatti

Dissertation presented in partial
fulfillment of the requirements for the
degree of Doctor in Engineering

July 2014

Dataflow-inspired parallel and distributed computing

Practical techniques and real-world use cases

Zubair Wadood BHATTI

Supervisory Committee:
Prof. dr. ir. Hendrik Van Brussel, chair
Prof. dr. ir. Yolande Berbers, supervisor
Prof. dr. Roel Wuyts, co-supervisor
Prof. dr. ir. Geert Deconinck
Prof. dr. Danny Huges
Prof. dr.ir. Tom Holvoet

Prof. dr. Wim Vanroose
(Universiteit Antwerpen)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

July 2014

Uitgegeven in eigen beheer, Zubair Wadood Bhatti, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

ISBN 978-94-6018-874-9

D/2014/7515/98

In the name of God, the Most Gracious, the Most Merciful

Acknowledgments

The research presented in this dissertation has only been possible with the help and support of many. I wish to acknowledge some of them.

I am grateful to my supervisors Yolande Berbers and Roel Wuyts for providing me with the support, help and guidance for accomplishing this research. I thank Yolande for providing me the opportunity to start my PhD at KU Leuven. She always encouraged me by being positive about my research throughout the span of the PhD. I learned not only research from her but also improved various soft skills under her guidance. I am thankful to Roel for always being available to guide me at every step and help me write papers. Roel always asked the right questions that helped me develop a deep understanding of the different topics in my thesis.

I am grateful to Geert Deconinck, Danny Huges, Wim Vanroose, Tom Holvoet and Hendrik Van Brussel for being a part of my PhD committee and for their insightful comments that helped significantly in improving this text.

I am indebted to Davy Preuveneers for always showing his sincere concern in my work. His advice and guidance has been fundamental to the progress of my research. Thank you Davy for proof reading most of my text and for always being very prompt with all kinds of help and support throughout my PhD.

I would like to thank all members of DistriNet. It was indeed a pleasure and an honor to be a part of this research group. I would like to thank especially the members of the embedded and ubiquitous systems task force for their suggestions and feedback in our meetings. Thank you,

Koosha Paridel, Naeem Muhammad, Ansar-Ul-Haque Yasar, Nayyab Zia Naqvi, Arun Ramakrishnan, Dries Langsweirdt, Aram Hovsepyan, Bert Vanhoof, Stefan Van Baelen and Yves Vanrompay. It was great to be a member of this team.

My sincere regards to the partners in the OptiMMA project. Thank you Narasinga Rao Miniskar, Shahid Mehmood Satti, Dirk Stroobant, Margarete Sackmann, Poona Behrabar, Leon Denis and Geert Vanmeerbeeck. It was delightful to work with all of you.

I would also like to acknowledge the members of the Intel ExaScience Lab for the technical/non-technical support and exchange of ideas. Thank you, Pascal Costanza, Albert-Jan Yzelman, Wim Heirman, Charlotte Herzeel, Gert Pauwels and everyone else. It was a great experience to collaborate with you.

Not forgetting the important contribution of my dear friends in Leuven. Thank you, Aftab, Amir, Ashraf, Ali, Asim, Azeem, Ansar, Zubair, Waqas, Hidayat, Pervaiz Baig, Sughis, Shahab, Shaza, Obaid, Nizabat, Ramzan, Ahmed Zaib, Bivragh, Khurram, Rehan, Zohaib, Muneeb, Salman, Amir Shah and Shakeeb for always being there for me. I would like to thank my office mates Italo and Andreas for their wonderful company.

Most importantly I would like to show my deepest gratitude to my family. I have no words to thank my dearest parents for their endless and sincere support throughout my life. Without their encouragement and guidance I would not have been able to achieve anything. I would like to thank my siblings Rabiya and Usman for their continuous concern and affection. My deepest gratitude to my lovely wife Zahra, not only for proof reading this text but also for her unconditional love and care throughout the journey. Her support in so many different forms was fundamental to the success of this thesis.

Abstract

Due to the advent of multi-processor system-on-chip (MPSoC), parallel and distributed computing has become one of the most active fields of research in computational science. Key challenges in parallel and distributed computing include: development of concurrent programming models, deployment, scheduling, memory management and synchronization. Dataflow inspired modeling and execution techniques have emerged as promising enabling technologies for addressing these concerns by organizing computation and data at a high level of abstraction.

While deemed promising, practical application of dataflow programming faces a number of challenges: (1) Unavailability of reusable libraries and kernels, (2) limitations of dataflow models in expressing dynamic data-dependent program structures, (3) inadequate middleware support and (4) lack of adaptions of existing techniques for different domains of parallel and distributed computing.

In this thesis, we show how these challenges can be overcome, not in a general way, but for specific applications, chosen from different application domains and with different characteristics. The first domain is high-performance computing, where we applied a dataflow-based programming model to stencil computations. The second domain is real-time embedded computing where we improved middleware support for dataflow models, focusing on optimizing schedules of applications on MPSoCs. The third domain is ubiquitous computing, where we apply dataflow-based modeling and optimization techniques, in order to deploy applications in ubiquitous environments.

Our results show that programming stencil operations with dynamic task graphs reduces their synchronization overhead, resulting in an implementation that scales better than the state-of-the-art. For mapping and scheduling applications on MPSoCs, we observe that modeling the inter-dependencies that enable tighter coupling between different aspects of the system i.e. computation, communication and memory, improves the energy efficiency and/or performance of the system. The execution semantics of dataflow models play a key role in enabling this tightly-coupled scheduling. Use cases in the domain of ubiquitous computing demonstrate the usage of dataflow-based modeling and optimization techniques for the deployment and configuration of dynamic applications in highly heterogeneous and distributed systems.

This dissertation showed how dataflow programming improves the programming of parallel and distributed systems for use cases in three different domains. We hope these are the first steps in the directions of a widely adopted dataflow-based renaissance in parallel and distributed computing.

Samenvatting

Als gevolg van de komst van multi-processor system-on-chip (MPSoC) is parallel en gedistribueerd rekenen uitgegroeid tot één van de meest actieve gebieden voor onderzoek in computerwetenschappen. De belangrijkste uitdagingen in parallel en gedistribueerd rekenen zijn het ontwikkelen van parallelle programmeermodellen en de implementatie, planning, geheugenbeheer en synchronizatie ervan. Een veelbelovende techniek om deze problemen aan te pakken zijn dataflow geïnspireerde modelleringsen uitvoer technieken.

Hoewel veelbelovend zijn er evenwel nog uitdagingen voor de praktische toepassing van dataflow programmering zoals (1) het niet beschikbaar zijn van herbruikbare bibliotheken, (2) de beperkingen in het uitdrukken van dynamische data-afhankelijkheden, (3) matige middleware ondersteuning, en (4) niet voldoende modellen die aangepast zijn aan de specifieke context van parallel en gedistribueerd rekenen.

Dit proefschrift laat zien hoe deze problemen kunnen worden aangepakt in de context van specifieke toepassingen. Deze toepassingen werden gekozen uit verschillende domeinen en hebben verschillende karakteristieken. Het eerste domein is high-performance computing, waar we een dynamisch dataflow-gebaseerd model gebruiken voor het uitvoeren van stencil berekeningen. Het tweede domain is real-time embedded computing waar we een verbeterde middleware ondersteuning voor dataflow modellen aanbieden gericht op het optimaliseren van toepassingen voor MPSoC platformen. Het derde domein is ubiquitous computing waar we dataflowgebaseerde modellering en optimalisaties gebruiken voor toepassingen in alomtegenwoordige omgevingen.

Onze resultaten met het gebruiken van dynamische dataflow voor stencil operaties laten zien dat we vooral de synchronisatie overhead beperken, wat resulteert in een betere performantie dan wat mogelijk is met de state-of-the-art stencil compilers. Wat het optimaliseren van toepassingen voor MPSoC systemen betreft blijkt dat onze oplossing die de drie belangrijkste deelaspecten van een applicatie (berekening, communicatie en geheugengebruik) samen modelleert een verbeterde energie-efficiëntie en/of prestatie van het systeem oplevert. Het is de uitvoerings-semantiek van dataflow modellering die een belangrijke rol speelt in het behalen van deze verbeterde resultaten. In het derde domein van alomtegenwoordige omgevingen demonstreren we het gebruik van dataflow-gebaseerde modellerings- en optimalisatietechnieken voor de implementatie en configuratie van dynamische toepassingen in zeer heterogene gedistribueerde systemen.

Dit proefschrift laat zien hoe dataflow-gebaseerde technieken het programmeren en uitvoeren van parallelle en gedistribueerde systemen verbetert in drie verschillende domeinen. We hopen dat dit de eerste stappen zijn in de richting van een algemeen aanvaarde dataflow-gebaseerde renaissance in parallelle en gedistribueerd rekenen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Almost half a century ago, Gordon Moore predicted [94] the exponential growth of the number of transistors on integrated circuits (IC) and he is proving to be almost[1] right, thanks to transistor scaling. As transistors shrank in size, it not only allowed more transistors to be packaged on a chip but also faster switching and thus higher frequencies for microprocessors. Transistor and frequency scaling combined with some architectural gizmos such as, instruction level parallelism (ILP), deep pipelining and out-of-order execution (OOO), resulted in a free lunch of performance for the software community. Unfortunately, this free lunch is now over [124].

The single core performance of the Von Neumann architecture based microprocessors hit several walls and hardware vendors now move towards multi-core parallel and distributed platforms. Concurrency becomes the primary way to get more performance on these platforms. However, concurrent programming is more difficult compared to sequential programming [125]. In this thesis we employ dataflow-inspired techniques to address certain challenges of parallel and distributed computing for some real world applications.

The rest of this chapter is structured as follows. Section 1.1 lists the

[1]The International Technology Roadmap for Semiconductors report for 2011 predicts the growth slowing down beyond 2013. It expects the number of transistors per IC to double every three years instead of two years, as predicted by Moore's law.

key factors driving the multi-core trend. Section 1.2 describes some challenges for the development and execution of parallel software, with respect to these multi-core platforms. Section 1.3 presents an overview of existing dataflow-based solutions to address these challenges. Section 1.4 introduces the use cases studied in this thesis. Section 1.5 gives a quick overview of the contributions of this PhD and section 1.6 presents the structure of the thesis.

1.1 Stagnation of the single core performance

The thesis focuses on software challenges of parallel and distributed computing. A major driving force behind the recent wave of parallel and distributed computing is the stagnation of the single core performance of microprocessors. This section highlights some aspects of the evolution of microprocessors that lead to the eventual slow down of the single core performance. Three problems that stall single-core performance scaling are: (1) The frequency wall, energy dissipation and thermal problems. (2) The tapered scaling of architectural features, such as ILP and pipelining. (3) The design complexity wall, i.e. the gap between technology scaling and hardware design productivity.

1. Power and energy dissipation have emerged as the Achilles heel for computing in several different domains. In embedded systems, it results in shorter battery lives of ubiquitous and mobile devices. In high performance computing, it manifests in the form of excessive electricity bills for datacenters. The power dissipation of a microprocessor increases super-linearly with the frequency [29], thus making it unfeasible to further increase the frequency.

   Thermal limitations are another reason why the frequency may not be further increased and seem to saturate below 4GHz for current semiconductor technologies. On a chip level, we are already reaching the limits of the amount of heat extractable from the chip using conventional methods of cooling. At the level of datacenters, the electricity cost of cooling the datacenters already exceeds that of the actual computation.

2. Even though the frequency may not increase, transistor scaling still continues exponentially. We may still design bigger processors, with more instruction level parallelism and deeper pipelines. Unfortunately, performance improvements with increasing ILP are observed to saturate at less ten instructions per cycle [124] and deeper pipelines reduce the energy efficiency of the processor [30].

   The increase of single core performance with respect to transistor scaling is summarized by **Pollacks's rule** [103] whereby, performance increases (when not limited by other parts of the system) as the square root of the number of transistors or area of a processor. According to Pollack's Rule, each new technology generation doubles the number of transistors on a chip, enabling a new microarchitecture that delivers a 40% performance increase.

3. Design and verification of a modern day microprocessor requires 3-5 years while hundreds of engineers are employed for the project. Historically the productivity of digital hardware design has increased at 21% per year (measured in transistors per staff months), whereas, the number of transistor per chip has grown at a much faster rate of 58% per year. This leaves an unmanageable design complexity gap [60, 100] between increasing number of transistors and productivity; leaving the vendors no choice but to replicate design in the form of multi-cores.

This section briefly summarized some of the reasons that drive the proliferation of multi-cores in different computing domains. This hardware trend has implications for software, which now needs to be parallel. However, developing parallel software or parallelizing existing ones are non-trivial tasks. The next section describes some of the challenges of developing and executing parallel and distributed software.

1.2 Challenges of parallel and distributed computing

This section highlights some important software challenges associated with parallel and distributed computing. Figure 1.1 gives a quick overview

of these challenges and illustrates that these challenges are deeply connected to each other. The programming model provides abstractions for synchronization and memory management. Synchronization is an intrinsic part of parallel programming models, from one perspective. Whereas, it is also a functionality of the scheduler, from another perspective. Similarly, memory allocation, deployment and scheduling decisions depend on each other. We will now discuss each of these challenges.

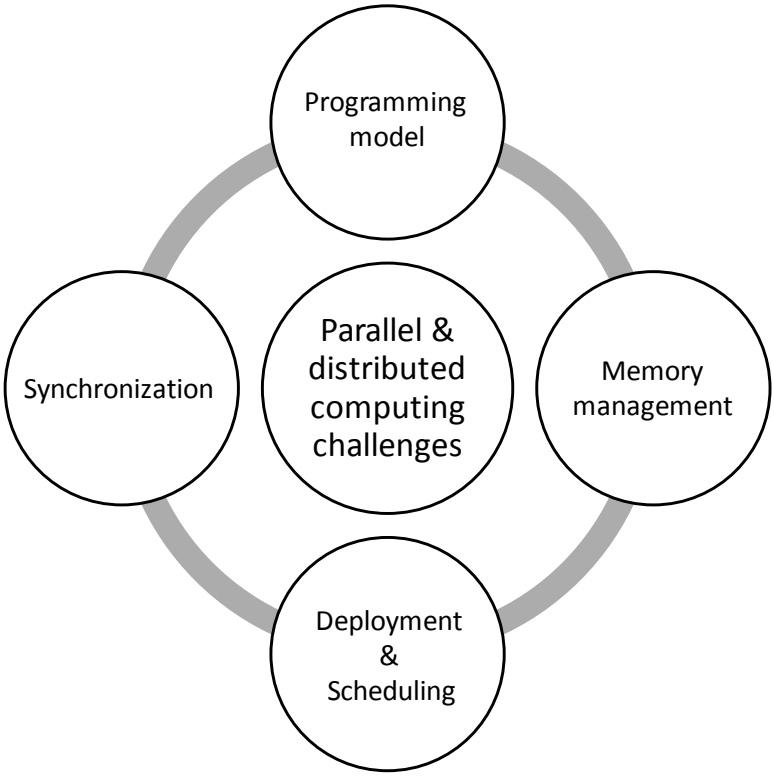


Figure 1.1: Some key challenges for parallel and distributed computing

1.2.1 Programming models

From the early days of *Al-Khwarizmi*, algorithms have been formulated with sequential semantics. Perhaps one of the biggest challenges of parallel programming is developing models and languages that allow developers to think about concurrency in a natural way. With the increasing parallelism in the underlying hardware, there is a need to expose higher levels of concurrency in the software applications. For many applications higher levels of concurrency require combining different forms of implicit and explicit parallelism, e.g. combining explicit functional parallelism with implicit data parallelism. Moreover, the programming model must guarantee functional correctness without over constraining the available parallelism, e.g. guarantee a race condition free execution. Furthermore, it must support a broad range of platforms, ranging from symmetric multi-processors with uniform memory access to heterogeneous multi-cores with non-uniform memory access. In addition, the programming model must allow the developer to reason about concurrency at a high level of abstraction. All these requirements make developing a parallel programming model an extremely difficult task.

Thread based programming models, such as POSIX threads [33] and OpenMP [42], have been considered the de-facto standard in parallel programming. More recently, task based programming models, such as Cilk [27], Threading Building Blocks [108] and OpenMP tasks [10] are gaining popularity. All of these programming models are control flow-based i.e. the programmer specifies the control flow of the parallel program. Different forms of concurrency have to be explicitly expressed and the programming models do not guarantee race condition free execution i.e. they are non-determinate. Section 1.3.1 discusses how dataflow-based programming models address these challenges.

1.2.2 Memory management

On the one hand, the "memory wall" [92] (i.e. bottleneck caused by the lack of memory bandwidth) is often considered a killer for the scalability of high performance parallel applications; if an application is starved of memory bandwidth adding more cores causes a performance degradation

due to contention on the shared resource. On the other hand, the memory subsystem dissipates a major chunk of the battery power in hand-held devices for data intensive applications (such as multimedia) [134]. One solution to these problems is using a multi-level memory subsystem that places fast and energy efficient caches and/or scratchpads [13] close to the processing cores. However, these caches and scratchpads are often limited in size and must be efficiently utilized in order to overcome memory bandwidth limitations in data intensive applications. Most existing cache and scratchpad allocation techniques [9, 115] for control flow-based programming models, often work in isolation from other aspects of the system i.e. processing and communication. Section 1.3.2 discusses how dataflow-based models help improve memory management techniques for parallel programs.

1.2.3 Scheduling

Optimal scheduling of parallel applications on parallel platforms is an NP-complete problem [43]. Scheduling is a crosscutting concern with aspects that often conflict with each other. For example, balanced workload distribution improves processor utilization but at the same time it usually puts more pressure on the communication network. In a heterogeneous system, particular types of workloads perform better on particular types of processors (e.g. CPU, GPU, DSP) and this type of affinity must be taken into account during the scheduling process. Similarly, the scheduler must also maintain locality for effective cache utilization. Furthermore, the algorithms/heuristics used must be scalable and the runtime overhead minimal. The analyzability of dataflow models (discussed in section 2.2) facilitates the development of efficient and comprehensive scheduling techniques.

1.2.4 Synchronization

Threads of a parallel application often need to synchronize due to dependencies, typically data or control flow dependencies. Synchronization points (e.g. locks and barriers) are necessary to avoid race conditions. Two important challenges are associated with synchronization;

(1) verifying that all the necessary synchronization points have been inserted and (2) the cost of synchronization. It is difficult to verify that all the necessary synchronization points are inserted because threads interact differently on different systems [84] and a race free execution on one system does not guarantee correctness on another. The cost of synchronization keeps rising with the increasing amount of parallelism and has a direct trade-off with the level of abstraction of the synchronization mechanisms. Control flow-based programming models such as Pthreads [33], OpenMP [42], TBB [108] and Cilk [27] provide two types of synchronization mechanisms; high-level all-to-all barriers that are expensive and low-level locks and semaphores that substantially increase the complexity of programming. Section 1.3.4 discusses the synchronization of dataflow programs.

1.3 Dataflow-based approaches to concurrent software

The previous section highlighted some key challenges in parallel and distributed software. We use dataflow-inspired techniques to organize computation and data at a high-level of abstraction, in order to address them. This section introduces dataflow models, discussing how they approach these challenges.

1.3.1 Dataflow programming models

In imperative (or control flow) styles of programming, programs are represented as sequences of operations. This is in line with the Von Neumann vision of computing. However, due to the inherent sequential nature of these programming models concurrency has to be expressed explicitly. In dataflow programming languages, a program is comprised of functions (nodes) and the flow of data between these functions (edges), where any set of functions may be executed in parallel as long as the dataflow dependencies are respected. Therefore, concurrency is an intrinsic property of the dataflow programming model.

Dataflow programming languages are classified according to the semantics of their models of computation; directed acyclic graphs (task graphs and dynamic task graphs [73]) or auto-concurrent models (e.g. synchronous dataflow graphs [85] and the actor model [3]). The edges in directed acyclic graphs are not allowed to form cycles and the nodes are executed only once during the execution of the program. DAG based programming models are often used for high performance computing applications e.g. in QUARK [145], SMPSs [101] and StarPU [8]. Whereas, in auto-concurrent models, the edges are allowed to form cycles and the nodes may be executed multiple times. Auto-concurrent programming models are more often used for embedded systems and real-time streaming application e.g. in StreamIt [128], brook [31], OpenDF [23], Caltrop [48], Erlang [7].

1.3.2 Dataflow-based memory management of concurrent programs

Pure dataflow programming models are characterized by ordered queues of data along the edges, single assignment variables and the absence of global memory store, thus, there are no side effects and the execution is determinate[2]. In dataflow graphs, nodes only process data locally and communication between nodes occurs only via the edges explicitly. This *locality of effects property* of dataflow programming models has been used to devise memory allocation schemes [16, 17, 34, 35] or to relax cache consistency models [116].

It is not possible to guarantee that an arbitrary dataflow graph can execute in a limited amount of memory. The sizes of the buffers that store the data flowing along the edges of the dataflow graph depends on the firing sequence of the nodes and hence the scheduler. A variant of dataflow graphs that specifies the amount of data produced and consumed at design time to make them statically schedulable is proposed in [83]. The requirements of scheduling more generic Kahn process networks in bounded memory are discussed in [58]

[2]The execution of dataflow programs in not necessarily deterministic but always determinate i.e. the sequence of how the nodes are fired may vary but the end result is always the same.

1.3.3 Dataflow-based scheduling of concurrent programs

In pure dataflow semantics, functions are scheduled on the basis of the availability of data. However, when executing dataflow programs on Von Neumann machines the availability of resources (such as cores for processing) also need to be considered alongside the logical availability of data. A number of scheduling algorithms and heuristics exist for different dataflow models, ranging from centralized static [142] and quasi-static [22] scheduling to distributed dynamic scheduling [4]; each having their own merits and de-merits (discussed in detail in section 2.2).

1.3.4 Synchronization of Dataflow programs

Synchronization in dataflow programming is implicit; the edges that represent dataflow abstract away the need for explicit synchronization. However, the implementation of these edges depends on the type of scheduler used. For statically scheduled dataflow programs, the edges may be implemented using asynchronous forms of communication. However, for dynamically scheduled dataflow programs, the implementation of these edges requires active synchronization (e.g. locks or mutexes).

1.4 Use cases

In this thesis we apply dataflow-inspired parallel and distributed computing techniques to use cases in three different domains: (1) High-performance computing, (2) real-time embedded software and (3) ubiquitous computing. This section describes the use case applications and the challenges addressed in each of these applications.

1.4.1 Stencil operations in high-performance computing

Stencil operations are at the heart of many high-performance computing (HPC) applications; such as, computational fluid dynamics (CFD) simulations, seismic simulations, plasma physics and partial differential equation (PDE) solvers. They often constitute a significant portion of

the computational time for these applications, therefore optimizing them usually has a big impact on the overall performance of these applications. With the ever-increasing number of on-chip cores in HPC servers, the scalability of parallel stencil operations is getting more and more important. Scalability of conventional parallel implementations of stencil operations face two important challenges: (1) Reducing synchronization overhead and (2) overcoming memory bandwidth bottleneck. We present an approach for addressing these challenges, thereby improving the scalability of parallel stencil operations.

1.4.2 Real-time applications in embedded systems

Modern embedded systems such as smartphones, tablets and other battery powered devices are often based on multi-processor systems-on-chip (MPSoCs) platforms with complex memory hierarchies and interconnects. Real-time applications that run on these devices often process large amounts of data with strict time constraints. Energy efficient deployment and scheduling of these software applications on complex hardware platforms taking into account the timing aspects is a daunting task. We present a technique for improving this deployment and scheduling, and validate it using two real life use cases: (1) Cavity detector (an image processing application) and (2) H.264 decoder (a video decompression application).

1.4.3 Component based applications in ubiquitous environments

Ubiquitous computing (ubicomp) is an advanced computing concept where computing is pervasive and deeply embedded into everyday objects. Propelled by decreasing costs and sizes of microprocessors, it is becoming the next big wave in the world of computing. Component based applications are often used in ubiquitous computing, where components are distributed in a ubiquitous environment and collaborate with each other to realize an application. Ubiquitous computing environments are a heterogeneous network of things, with different sensing, actuating, computing and communication capabilities. Moreover,

they are characterized by an open-ended and highly dynamic ecosystem with variable workload and resource availability, which adds to the complexity of deployment and configuration of application components in these environments. We present a methodology for the deployment configuration and runtime reconfiguration of application components in ubiquitous environments and validate it with two use cases: (1) Energy-aware application deployment of software applications in wireless sensor networks (WSNs) based on the *Component and Policy Infrastructure* CaPI [91]. (2) An ambient assisted living application that monitors fitness by recording activity and detects falls for people with special needs in order to inform care givers in case of an emergency.

1.5 Contributions

The previous section introduced the use cases studied in this thesis and discussed some challenges for each of these use cases. In this section we highlight the main contributions of the dissertation.

1.5.1 Contribution 1: Parallel stencil operation using dynamic task graphs

A stencil operation iteratively updates elements of an array using values of other elements (often the neighboring elements) of the array. Parallel versions of these stencil operations often suffer from over-synchronization and memory bandwidth saturation. We use *dynamic task graphs* (a dataflow-based programming model) to parallelize stencil operations in order to reduce the synchronization overhead. Moreover, *time-tiling* (an advanced algorithmic transformation) is used to reduce cache misses and avoid memory bandwidth saturation. Performance benchmarking results on a 16 core parallel system indicate performance improvements of at least 20% compared to the state-of-the-art heavily optimized stencil compilers while the synchronization costs are reduced by at least a factor of five.

Related publication:

- Zubair Wadood Bhatti, Roel Wuyts, Pascal Costanza, Davy Preuveneers, Yolande Berbers, *"Efficient Synchronization for Stencil Computations Using Dynamic Task Graphs,"* Elsevier Procedia Computer Science, special issue on *International Conference on Computational Science*. Volume 18, 2013, ISSN 1877-0509.

1.5.2 Contribution 2: Design-time exploration of Pareto optimal mappings for embedded applications

Scheduling and executing software efficiently on contemporary embedded systems, featuring heterogeneous multiprocessors, multiple power modes, complex memory hierarchies and advanced interconnects, is a daunting task. State-of-the-art tools that schedule software tasks to hardware resources face limitations: (1) either they do not take into account the inter-dependencies among processing, memory and communication constraints (2) or they decouple the problem of spatial assignment from temporal scheduling. As a result existing tools make sub-optimal spatio-temporal scheduling decisions. This dissertation presents a technique to find globally optimized solutions by co-exploring spatio-temporal schedules for computation, data storage and communication simultaneously, considering the inter-dependencies between them. Case studies of mapping image processing applications on a heterogeneous MPSoC platform show that this co-exploration methodology finds schedules that are more energy efficient when compared to decoupled exploration techniques for the particular application and target platform.

Related publications:

- Zubair Wadood Bhatti, Narasinga Rao Miniskar, Roel Wuyts, Davy Preuveneers, Yolande Berbers, *"SAMOSA: Scratchpad aware mapping of streaming applications,"* Proceedings of the 13th International Symposium on System-on-Chip, IEEE, pp.48,55, Oct. 31 2011-Nov. 2 2011, Tampere, Finland.

- Zubair Wadood Bhatti, Narasinga Rao Miniskar, Roel Wuyts, Davy Preuveneers, Yolande Berbers, Francky Catthoor, *"Memory and communication driven spatio-temporal scheduling on MPSoCs,"* Proceedings of the 25th Symposium on Integrated Circuits and Systems Design, IEEE, pp.1,6, Aug. 30 2012-Sept. 2 2012, Brasilia, Brazil.

1.5.3 Contribution 3: Exploring deployment and configuration tradeoffs for ubiquitous computing applications

We present a methodology for the deployment, configuration and runtime reconfiguration of component based ubiquitous computing applications. Results from two use cases are presented: (1) A component and policy infrastructure based WSN and (2) Mobility monitoring and fall detection application. The first use case results demonstrate the ability of the approach to find optimal deployment configurations for different scenarios and topologies in a distributed and heterogeneous system. The second use case takes a step further by relaxing the fixed quality of service constraints and exploring the trade-offs between quality of service and CPU load.

Related publications:

- Zubair Wadood Bhatti, Nayyab Naqvi, Arun Ramakrishnan, Davy Preuveneers and Yolande Berbers, *"Learning Distributed Deployment and Configuration Trade-offs for Context-Aware Applications in Intelligent Environments"* to appear in Journal of Ambient Intelligence and Smart Environments, IOS press, Volume 6, 2014.
- Arun Ramakrishnan, Syeda Nayyab Zia Naqvi, Zubair Wadood Bhatti, Davy Preuveneers, Yolande Berbers, *"Learning deployment trade-offs for self-optimization of Internet of Things applications"*, Proceedings of the 10th International Conference on Autonomic Computing, ICAC 2013, pages 213-224, San Jose, CA, U.S.A., 26-28 June 2013.

- Arun Ramakrishnan, Zubair Wadood Bhatti, Davy Preuveneers, Yolande Berbers, Aliaksei Andrushevich, Rolf Kistler, Alexander Klapproth, *"Behavior modeling and recognition methods to facilitate transitions between application-specific personalized assistance systems"*, International Joint Conference on Ambient Intelligence, Pisa, Italy, 13-15 November 2012.

1.6 Structure of the thesis

The rest of the thesis is structured as follows: Chapter 2 gives background information on dataflow programming models and their execution. Chapter 3 describes the first contribution, using a dataflow programing model for better parallelization of stencil operations in the area of high performance computing, reducing the synchronization costs. Chapter 4 presents the second contribution, improving the mapping and scheduling of real-time streaming applications by modeling the inter-dependencies between the different aspects. Chapter 5 explains the third contribution of exploring non-functional deployment trade-offs for component based applications in ubiquitous environments. Chapter 6 presents the conclusions and future work.

Chapter 2

Parallel programming using dataflow models

The previous chapter gave an overview of the dissertation, which uses dataflow-based programming models to address some key challenges in parallel programming. This chapter provides background on dataflow models and their execution on parallel Von Neumann computers. The first section presents a taxonomy of dataflow programming models. The second section discusses different scheduling techniques used for executing dataflow programming models on parallel computers. The third section gives an overview of memory management; discussing buffer dimensioning methodologies, optimizations for improved cache utilization and scratchpad allocation techniques. The last section highlights the synchronization of dataflow models.

2.1 Dataflow programming models

Dataflow programming models are a class of concurrent programming models. In contrast to imperative programming models that focus on the control flow of the program, dataflow programming models focus on the dataflow of the program. In dataflow programming models, the program is represented as a directed graph. The nodes of the graph represent the

computation and the edges represent the flow of data between these nodes. Dataflow models are implicitly parallel: the nodes in dataflow programs can be executed in parallel as long as they have the required data on their input edges. To structure the discussion of different dataflow models we made a taxonomy, illustrated in figure 2.1. At the top level dataflow models are classified by the type of their nodes i.e. tasks or processes. Tasks execute once during the span of the program, whereas, processes may execute (fire) multiple times. Sections 2.1.1 and 2.1.2 describe the task and process based models respectively.

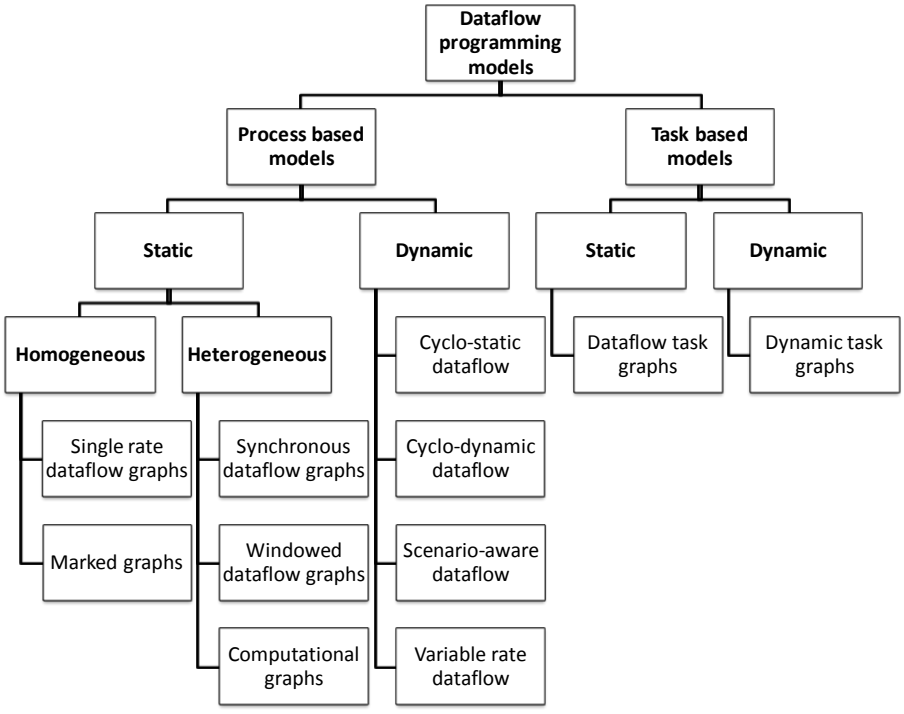


Figure 2.1: Taxonomy of dataflow programming models

2.1.1 Task based dataflow models

Task based dataflow models are structurally acyclic; an arbitrary pair of tasks may have a dependency between them as long as they do not form cycles. These models are further classified as static or dynamic. For static task graphs, the structure of the graphs does not change at runtime and new nodes or edges may not be created once the execution of the program starts. In contrast, dynamic task graphs [73] allow creation or deletion of nodes and edges at runtime. Both static and dynamic task graphs are often found in high performance computing applications e.g. QUARK [145], StarPU [8] SMPSs [101] and Nabbit [4].

Dataflow-based task models are conceptually different from the task graph models in imperative programming that impose a tree structure for the graphs, e.g. spawn-sync graphs in Cilk [27] or fork/join graphs in TBB [108] or java. These imperative task graphs only support parent-child dependencies between tasks; unlike dataflow task models arbitrary dependencies cannot be directly expressed.

2.1.2 Process based dataflow models

In process based dataflow programming models, the program may contain cycles. The processes may fire (execute) multiple times during the span of the program, consuming data from the input edges and producing data on the output edges each time. In figure 2.1, these models are classified as static or dynamic models.

In static dataflow graphs, the amount of data produced and/or consumed during a single execution of a process (data rate) remains static throughout the execution of the program. These models have two sub-classes; homogeneous [95] and heterogeneous. In homogeneous models [40, 95], all processes have the same data rates; whereas, in heterogeneous models [54, 83, 139] different processes have different data rates.

In dynamic dataflow graphs, the data rates are allowed to vary over time. In cyclo-static dataflow graphs[25], the data rates of different processes follow fixed periodic sequences. Cyclo-dynamic dataflow graphs [136] are an extension of cyclo-static dataflow graphs, where the sequence of

data rate is not fixed but rather a function of the given input(s). In scenario-aware dataflow graphs [120], the data rates of processes are controlled by a Markov model, where the states of the Markov model represent different runtime scenarios.

2.2 Scheduling of dataflow programs

Theoretically, processes or tasks of a dataflow program are executed as soon as the required data arrives at their incoming edges. In practice, other aspects are also taken into account while scheduling dataflow programs on Von Neumann computers, along with the execution semantics of dataflow models. First of all, hard constraints such as the availability of processing cores, memory and communication bandwidth need to be ensured. Secondly, due to the often-large number of feasible possibilities of deploying and scheduling dataflow programs, optimization objectives such as maximizing throughput and/or minimizing energy dissipation are usually considered. A number of scheduling algorithms and heuristics exist for different dataflow models, ranging from static to quasi-static and dynamic. Table 2.1 summarizes what type of scheduling techniques are applicable for the different dataflow models shown in figure 2.1.

2.2.1 Static scheduling

Static (task or process based) dataflow models can be scheduled offline, statically. Lee and Messerschmitt [85] proposed techniques for scheduling synchronous dataflow graphs (SDFs) onto single and multiple processors. These techniques find *periodically admissible sequential schedules* (PASS) and *periodically admissible parallel schedules* (PAPS). First, a *topology matrix* for the program is constructed from the SDF model. In a topology matrix, the columns represent the processes and the rows represent the edges; an entry at row r and column c represents the number of tokens (data objects) produced (positive number) or consumed (negative number) by process c on edge r. A PASS only exists if the rank of the topology matrix is one less than the number of rows in the matrix. Otherwise,

the program is unschedulable. Calculating the PASS involves finding the firing vector $\vec{q}$; such that,

$$\vec{q} \times A = \vec{0}$$

where A is the topology matrix of the SDF and the elements of $\vec{q}$ are positive integers. If a PASS exists, PAPS can always be computed with the following steps:

1. Compute a PASS.
2. Determine the *unroll factor* that is the number of PASS that forms a PAPS. Higher unroll factors normally improve the utilization in a multi-processor system at the cost of increased complexity.
3. Construct a precedence graph.
4. Compute PAPS using the Hu-Level algorithm [69].

Static scheduling is done offline before the application starts. Thus, there is minimum overhead at runtime and complex optimizations that are unfeasible for purely runtime scheduling techniques can be performed. However, static scheduling techniques lack in their ability to cope with dynamism. Often, worst-case estimates are taken in order to deal with dynamism in static dataflow models. These worst-case estimates are sometimes too conservative and cause severe performance penalties. Therefore, dynamic dataflow models are used. However, these dynamic dataflow models cannot be scheduled with static scheduling techniques; instead, quasi-static [22], scenario-aware [62] or fully dynamic scheduling techniques are used.

2.2.2 Dynamic scheduling

Dynamic schedulers make all the scheduling decisions at runtime. The key advantage of these schedulers over their static counterparts is the ability to cope with dynamism. The main objectives for dynamic schedulers include meeting timing deadlines, fulfilling quality of service requirements, maximizing system utilization, preserving locality and balancing workloads etc. Different types of dynamic schedulers are

found in the literature, e.g., deadline driven schedulers, priority driven schedulers and best effort schedulers.

- ***Deadline driven*** scheduling is often used for real-time systems. The most common deadline driven schedulers are *Earliest Deadline First* (EDF) [71] and *Least Slack First* (LST) [1]. In EDF, the tasks or processes with the closest deadline are scheduled first. Similarly, in LST the tasks or processes with the least slack time are scheduled first. Slack time is the difference between the time remaining before deadline and the time required to finish the task or process. These schedulers work with both periodic and sporadic tasks.

- ***Priority driven*** scheduling is used in interrupt driven systems and operating systems for embedded or real-time systems, such as the *Rate Monotonic Scheduler* (RMS) [86] in microC/OS. In priority driven schedulers, tasks (or processes) are assigned predefined priorities by the application developers or by the system architects. A task is only allowed to execute if it has the highest priority among all ready tasks. The disadvantage of both priority driven and deadline driven schedulers is the often-low system utilization level (e.g. as low as 69% for RMS [86]).

- ***Best effort*** scheduling is more commonly used in high performance computing. The most common best effort schedulers include *work-sharing* schedulers (e.g. in OpenMP [42]) and *work-stealing* schedulers (e.g. in TBB [108] and cilk [27]). Best effort schedulers generally provide better system utilization, workload balancing and locality awareness than deadline or priority driven; however, they do not provide guarantees for meeting specific task deadlines. In work sharing schedulers, when processors create new tasks they try to assign them to underutilized processors. In work stealing schedulers, when a processor runs out of tasks it steals a task randomly from another processor.

2.2.3 Quasi-static and scenario-aware scheduling

Quasi-static [22] and scenario-aware [62] scheduling techniques strive for a middle ground between purely static and purely dynamic techniques; benefiting from the advantages of static scheduling whilst being able to deal with some forms of dynamism. Cyclo-static [25], cyclo-dynamic [136] and scenario-aware [120] dataflow graphs that express limited form of dynamism can be scheduled with quasi-static or scenario-aware scheduling.

In scenario aware deployment and scheduling of dataflow graphs, static schedules are constructed offline for a number of possible runtime scenarios. These pre-computed schedules are then used according to the situations, at runtime. The runtime situations are continuously monitored and schedules are switched if necessary or if beneficial. Thus, by doing most of the computation related to scheduling at runtime the overhead of scheduling is kept low, whilst dynamism is supported in the form of scenarios.

Dataflow model	**Static sch.**	**Quasi-static sch.**	**Dynamic sch.**	**Buffer size analysis**
Single rate dataflow	✔	✔	✔	✔
Marked graphs	✔	✔	✔	✔
Synchronous dataflow	✔	✔	✔	✔
Windowed dataflow	✔	✔	✔	✔
Computational graphs	✔	✔	✔	✔
Cyclo-static dataflow	✘	✔	✔	✔
Cyclo-dynamic dataflow	✘	✔	✔	✔
Scenario-aware dataflow	✘	✔	✔	✔
Variable rate dataflow	✘	✘	✔	✘
Dataflow taskgraphs	✔	✔	✔	✔
Dynamic taskgraphs	✘	✘	✔	✘

Table 2.1: Schedulability and buffer size analysis for different dataflow models

2.3 Memory management of dataflow programs

Modern computing systems consist of complex memory hierarchies, e.g. in the form of multi-level caches and scratchpads in MPSoCs, and ccNUMA (cache coherent Non-Uniform Memory Access) [80] interfaces for random access memories (RAMs) in multi-socket compute nodes. Caches and scratchpads are smaller and faster memories, near the processor. Caches are managed by a dedicated hardware unit the cache controller, whereas, scratchpads are software managed. In multi-socket compute nodes, the main memory is logically shared but physically distributed. The bandwidth and latency of a data access depends on the physical location of the data. Efficient execution dataflow programs on these systems (MPSoCs or compute nodes) requires that the memory subsystem is taken into account. Firstly, the buffers required to execute the program must be smaller than the available memory in order to avoid deadlocks and ensure correctness. Secondly, caches and scratchpads must be efficiently utilized in order to get maximum throughput and energy efficiency. The remainder of this section gives an overview of buffer dimensioning techniques for dataflow graphs followed by cache optimizations and scratchpad allocation techniques.

2.3.1 Buffer dimensioning

Sizes of the memory buffers required to execute a dataflow program depend on its schedule [58]. Table 2.1 shows which dataflow models can be analyzed for buffer size requirements. For statically or quasi-statically scheduled dataflow programs, it is possible to calculate the buffer sizes by analyzing the schedules. Model checkers and constraint based optimization tools are often used to calculate the buffer requirements for these dataflow models. In [57] a heuristic is presented for the calculation of minimum buffer requirements for which a schedule can exist that executes an SDF program without deadlocking. The operational semantics of the SDF and arbitrary channel bounds are encoded in a model checker, the model checker is then challenged to disprove the claim that a schedule that fulfills all the constraints does not exists. If the model checker disproves the claim, the memory limit is lowered to find the minimum

buffer sizes for which a schedule exist. This technique is extended in [121] for exploring tradeoffs between buffer sizes and throughput. An analytical technique for approximating buffer requirements for a given throughput constraint is presented in [138].

2.3.2 Cache optimization techniques

From a software perspective, caches transparently store data, so that future requests for this data are served faster. In essence they rely on the notion of spatial and temporal locality; a data object accessed once has a higher probability of being accessed again and that other data objects nearby are also likely to be accessed in the near future. Therefore, increasing the spatio-temporal locality of data references of a program usually improves the caching performance. Two types of optimization techniques are commonly employed in order to improve the spatio-temporal locality of programs: (1) Data access optimizations and (2) data layout optimizations.

1. **Data access optimizations** are code transformations that change the order of data accesses. These include basic loop transformations for perfectly nested loops[1], e.g., *loop interchange, loop fusion, loop fission* and *loop tiling*. However, real-life programs are often not perfectly nested and certain pre-conditioning transformations are usually required [76]. Examples of such pre-conditioning transformations are *loop skewing, loop unrolling* and *loop peeling*.

2. **Data layout optimizations** rearrange data in the memory in order to improve spatial locality and/or reduce cache conflicts and false sharing [129]. A common data layout transformation is *array padding*. When data elements mapped onto the same cache line are accessed in an alternating fashion, they cause cache conflicts. Array padding adds unused variables (pads) between the data elements in order to map them to different cache lines. Similarly, *data coping* [141] is used to copy data from non-contiguous memory locations to contiguous areas of memory, thus improving caching

[1] Perfectly nested loops are nested loops where all assignment statements are contained in the innermost loop.

performance. Other examples of data layout transformations include *array merging* and *array transpose*.

Beside these basic optimizations, other advanced domain specific transformations are used for specific applications. For example, in the domain of numerical applications, space filling curves (such as Peano curve, Hilbert curve and Z-order curve) are used to maximize spatial locality of multi-dimensional matrix multiplication [66]. Similarly, time tiling [53] is used to increase the arithmetic density of stencil computations much beyond conventional loop tiling. Thus memory bandwidth bottlenecks are avoided in these applications.

2.3.3 Scratchpad allocation techniques

The need for energy efficiency requires that the most frequently used data objects be kept in faster and more energy efficient memories. Up until the early 2000s, caches were considered the de facto standard for such memories. In [14], it is shown that scratchpad memories generally consume 40% less energy and 34% less area as compared to caches of the same storage capacity. The effect on energy consumption of the system overall is however dependent on the efficiency of scratchpad management. Two of the biggest challenges in scratchpad allocation are data reuse analysis and dynamically changing the set of objects assigned to the scratchpad. Evolution of scratchpad allocation techniques is classified as follows:

1. ***Data objects types:***
   The initial techniques only considered static variables for scratchpad allocation. Later techniques started considering stacks as well [9, 115] and more recent techniques work with all variables including the ones that go on the heap [47].

2. ***Program structure requirements:***
   Early work required the code to be very well structured. For example, variables could only be accessed in the inner most loops and without control flow statements such as if-else and continue-

break [75, 110]. But recent techniques are more generalized to work with irregular control flow and non-affine access patterns [130].

3. ***Data-reuse analysis:***
   One way of analyzing data-reuse is through offline static analysis of the source code[9, 110], however, these techniques usually impose limitation on the structure and semantics of the application. Another way is through profiling [5] but this has limitations for dynamic applications.

4. ***Dynamism:***
   The set of most frequently used data objects may change over time. This requires that the data objects allocated to the scratchpad are also replaced over time. The challenge lies in evaluating the expected performance gains by allocating new objects to the scratchpad verses the cost of allocations/de-allocations [52, 75].

2.4 Synchronization of dataflow programs

Parallel programs that work with shared variables require synchronization at runtime if they are dynamically scheduled. Synchronization is primarily needed to avoid race conditions. A race condition [99] occurs when a parallel program produces an incorrect output due to violation of a read after write dependency on a shared variable. Pure dataflow models [25, 83, 120] do not allow shared variables and all communication between nodes is explicit. However, implementations of dataflow models on Von Neumann computers often use shared variables to realize communication on shared memory systems and therefore require synchronization.

Synchronization is usually defined by specifying synchronization points in the program. Mechanisms for specifying these synchronization points vary in their levels of abstraction, ranging from low-level locks and semaphores to high-level barriers (e.g. in OpenMP [42],Pthreads [33],TBB [70, 77, 108] and Cilk [27]). Low-level synchronization mechanisms, such as locks and semaphores have minimal overhead and provide point-to-point synchronization, thus are generally more efficient. However, this efficiency comes at the price of increased complexity. The complexity

of developing applications with low-level synchronization mechanisms increases not only with the scale of the application but also with the increasing concurrency. High-level synchronization mechanisms, such as barriers provide system wide all-to-all synchronization thus decrease the complexity of developing applications, however, they have a larger overhead and often cause over-synchronization.

2.5 Conclusion

This chapter provided background on parallel programming with dataflow models. It classifies different dataflow models in the form of a taxonomy and discusses the scheduling, memory management and synchronization for these models. The following chapters discuss the contributions of this dissertation in detail.

Chapter 3

Parallel stencil computations using dynamic task graphs

The previous chapter provided background on parallel programming using dataflow models. This chapter uses a dataflow programming model, *dynamic task graph* (section 2.1.1) to program stencil operations. Evaluation with two benchmark applications show that the dynamic task graph based stencil implementation has minimal synchronization overhead and scales better than the state-of-the-art implementations of the same kernel, demonstrating the competitive potential of dataflow models in the domain of high performance computing.

3.1 Introduction

Stencil operations

Stencil operations are a class of iterative methods which update array elements according to a fixed pattern, called *a stencil*. The array usually represents a 2 or 3 dimensional grid, where each element of the array represents a grid cell. A complete sweep applying a stencil operation over the whole grid is called a *timestep*. One of the primary applications of stencil operations in high-performance computing is for numerical

simulations. An example can be found in the field of computational fluid dynamics (CFD), where the interaction of liquids and gases with surfaces is simulated. This simulation is used to improve the aerodynamic properties of cars and planes, or to design swimsuits that reduce the drag in water and make swimmers like Michael Phelps swim faster. From a high-level perspective, CFD simulations approximate a continuous solution described by equations with a series (timesteps) of discretized solutions. Stencil operations calculate the new values (for the current timestep) of the discretized space based on the values of a previous timestep.

Figure 3.1 shows a two dimensional five-point stencil, commonly used in the Jacobi's iterative method for solving partial differential equations (PDEs), such as the heat equation or Laplace's equation. The function F() uses the green cells in timestep t to compute the value for the orange cell in timestep $t + 1$ and this operation is repeated for each cell in a timestep. Stencil computations have a lot of intrinsic data parallelism available within a single timestep, however, there are data dependencies between the different timesteps that need to be respected in order to produce correct results. In this example, each row in the grid is assigned to a thread, for the sake of simplicity (the actual implementations use block-based data distribution to minimize border communication). Since

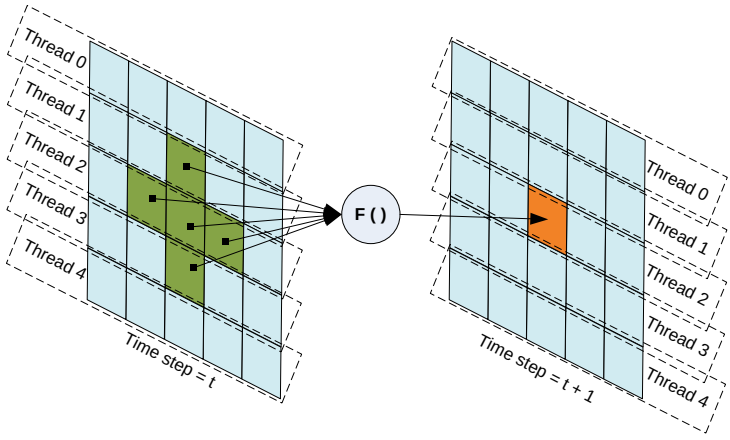


Figure 3.1: A five point stencil operation on a two dimensional grid

neighboring rows are assigned to different threads, threads must ascertain that the required input data is available before starting computation. For example, thread 2 in figure 3.1 must synchronize with threads 1 and 3 to make sure all the data required for its computation is ready. Stencil computations usually have a low arithmetic intensity (arithmetic operations per memory access), therefore, efficient use of the cache plays a pivotal role in the overall performance of the kernel.

Other examples of high-performance computing applications that rely heavily on stencil computations include: Simulation of plasma in nuclear physics or space research, weather simulations, shock hydrodynamics, combustion simulations etc. Most of these high performance computing applications involve solving linear systems of partial differential equations (PDEs) or non-linear systems of hyperbolic partial differential equations. Stencil operations lie at the heart of all these systems [45], whether they involve structured (rectangular) grid methods, multigrid methods [63] or block-structured adaptive mesh refinement based techniques [39].

Parallel stencil operations in high-performance computing

Over the past decade the computing industry has witnessed a shift from exponential frequency scaling to increasing number of cores on chip. This trend is expected to continue in the foreseeable future for various reasons, including energy efficiency, performance, reliability and processor design costs. It is projected that, during the next decade the amount of parallelism on a single microprocessor will rival that of the early supercomputers built in the 1980s [111]. This trend of increasing parallelism is seen in nearly all computing devices, ranging from smart-phones and tablets to multi-socket servers. Contemporary High Performance Computing (HPC) clusters are based on connecting individual *compute nodes* with fast (low-latency, high-bandwidth) interconnects, such as Infiniband networks. Each compute node usually has multiple sockets that house multi-core processors and additional co-processors (such as Intel Xeon Phi in Tianhe-2 [41]) or accelerator cards (such as NVidia Tesla in Titan). The result is a hardware setup with a complex memory hierarchy and communication topology.

The software applications that use stencil codes typically run on high-performance computing infrastructure and therefore have to deal with this complex hardware setups. In general, software for high-performance computing needs to combine parallel programming (on the compute nodes, that are shared-memory systems) and distributed programming (across compute nodes). While attempts are being made to develop programming models that span the complete systems (such as the Partitioned Global Address Space model [146]), no efficient generally applicable programming model exists at the time of writing. State-of-the-art software for HPC is therefore typically developed in a so-called *hybrid* fashion, where a shared memory parallel programming model (such as OpenMP or Threaded Building Blocks) is combined with a distributed programming model (such as MPI). This chapter focuses on the optimization of stencil computations on shared memory parallel platforms consisting of multiple sockets with multi-core processors.

The scalability of high performance computing applications becomes more and more important with the ever increasing number of cores per chip. Two problems that hamper the continued scalability of parallel stencil computations are: (1) Over-synchronization and (2) memory bandwidth saturation. Several *stencil compilers* and *auto-tuning frameworks* have been developed that focus on optimizing stencil computations on multi-core hardware, for example, Pochoir [126], Pluto [15], Patus [37] and the Berkeley auto-tuner [44, 45]. These stencil compilers and auto-tuning frameworks take a stencil description (written in a domain specific language) and generate optimized multi-threaded code. They use techniques like time-tiling [53] and cache blocking to increase the arithmetic intensity i.e. doing more arithmetic operations per memory access. While these techniques are very effective in increasing the arithmetic intensity, they use global barriers for the synchronization, which cause a lot of overhead. Techniques that propose low-level point-to-point synchronization for stencil computations using Phasers are presented in [112, 113]; section 3.8 discusses them in more detail.

Stencil operations using dynamic task graphs

We model stencil computations using dynamic task graphs (DTGs) [73], which are directed acyclic graphs dynamically created at runtime and typically executed using a work-stealing scheduler (section 2.1.1). By construction, this approach avoids global barriers, but uses point-to-point synchronization instead. This point-to-point synchronization is further reduced because of lazy task spawning i.e. delayed spawning of tasks until they can execute without the need for any synchronization. Furthermore we exploit basic properties of stencil computations for memory management and performance optimization of dynamic task graphs. Moreover, time-tiling is used to increase the arithmetic intensity of the stencil operations in order to reduce the cache misses and eliminate the memory bandwidth bottleneck. To the best of our knowledge this is the first dataflow implementation of stencil operations that employes time-tiling, resulting in fast and scalable stencil operations.

The approach is implemented using the *Flowgraph* library [135] included in the *Intel Threading Building Blocks* (TBB) version 4.1. The TBB Flowgraph library supports static task graphs with dataflow execution semantics. By adding support for dynamic task creation at runtime, we are able to model dynamic task graphs using Flowgraph objects. These dynamic task graphs are then executed using the TBB work-stealing scheduler. We use two benchmarks for validation and compare our results to the state-of-the art stencil compilers Pochoir [126] and Pluto [15].

The rest of the chapter is structured as follows. Section 3.2 highlights key problems in the scalability of parallel stencil operations. Section 3.3 explains dynamic task graphs and their execution. Section 3.4 presents the modeling of stencil operations as dynamic tasks graphs. Section 3.5 explains our implementation of dynamic task graphs using TBB Flowgraphs. Our experimental results are shown in section 3.6. Section 3.7 presents further discussions on some aspects of our approach, including synchronization, vectorization and non-uniform memory access (NUMA) awareness. Section 3.8 gives an overview of the state-of-the-art in the field of stencil computations. Section 3.9 presents our conclusions.

3.2 Scalability of stencil operations

This section describes in detail the problems with developing efficient stencil codes for multi-socket multi-core compute nodes. Figure 3.2(a) shows an implementation of the stencil operation of figure 3.1 parallelized using an OpenMP *parallel_for* construct. Figure 3.2(b) shows a thread model of the implementation, with a barrier between each timestep. Figure 3.2(c) shows the memory access pattern for the stencil as it moves in the increasing x direction. The performance scaling of this implementation on a dual socket Intel Xeon E5-2670 (2x8 cores) is shown in figure 3.2(d). With compiler based vectorization, pre-fetching and -O3 optimization for speed, the system performs only 23 GFLOPS[1] (23 billion floating point operations per second), which is less than 15% of the computational power of the node. Two reasons for this lack of computational efficiency are: (1) over-synchronization and (2) memory bandwidth saturation.

3.2.1 Over-synchronization

One of the major factor inhibiting the scalability of parallel software on multi-core systems is the synchronization overhead. Consider the parallel stencil implementation shown in figure 3.2(a). The top loop (using variable `t`) cannot be parallelized using *parallel_for* due to the dependencies between successive iterations, where a later iteration requires values of the previous iteration and therefore can only start when the previous one has finished. The second loop (using variable `y`) can be parallelized because its iterations can be executed independently of each other. However, the parallel_for end with barrier synchronization, thus by parallelizing the second loop a barrier is inserted between iterations of the top loop (timesteps).

Figure 3.2(b) shows a corresponding thread model of the implementation in figure 3.2(a), showing the barriers between successive timesteps. In this parallel version of the stencil operation neighboring rows are assigned to different threads. Before thread 2 starts computing timestep 1 it

[1]Similar implementations using Cilk Plus or Threading Building blocks have comparable performance.

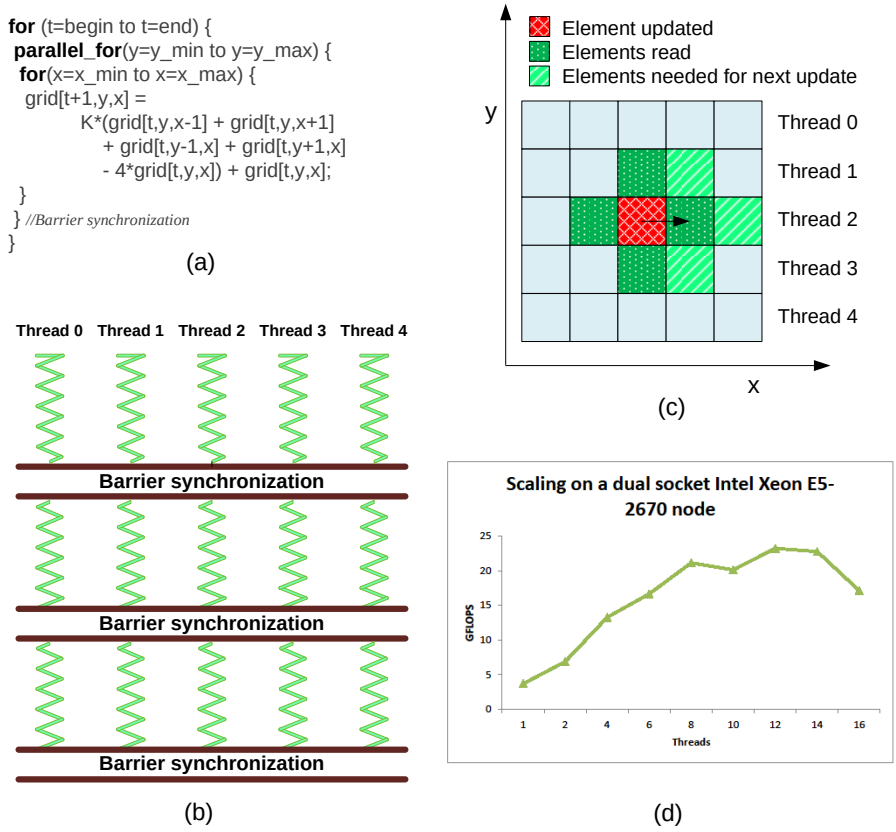


Figure 3.2: A naively parallelized 2D five-point stencil: (a) An OpenMP C++ implementation, (b) a thread model showing barrier synchronizations between subsequent timesteps, (c) memory access pattern and (d) performance scaling results on a 16 core system.

must make sure that threads 1 and 3 are finished with timestep 0, in order to avoid a race condition. This synchronization is provided by an implicit barrier at the end of the *parallel_for* loop. Note that thread 2 only required synchronization with threads 1 and 3 but due to the "all-to-all" nature of the barrier it is forced to synchronize needlessly with threads 0 and 4 as well, thus causing over synchronization. Similar over-synchronization problems are even found in the state-of-the-art

implementations of stencil operations, as we will show in Section 3.7.1. The next section shows how our approach minimizes the synchronization overhead by using the execution semantics of a dynamic task graph.

3.2.2 Memory bandwidth saturation

A second important factor inhibiting the scalability of stencil operations is memory bandwidth saturation. Naive implementations of stencil operations are usually unable to exploit the computational power of the platform due to the bottleneck caused by bandwidth saturation. Consider the stencil operation in figure 3.2. Assuming that the domain of the stencil (e.g. the grid on which the stencil operation is performed) is much larger than the cache, every lattice update requires (apart from the boundaries) 7 floating point operations for 32 bytes of data transfer (3 reads and 1 write) to/from the memory, resulting in a low 0.219 FLOPS/byte ratio. To improve this so-called *cache-aware loop tiling techniques* we divide the domain into tiles such that two complete rows of the tile may be cached. This reduces the main memory traffic down to a single read and write, resulting in an overall 0.438 FLOPS/byte. However this is still lower than what most modern processors can achieve, which has historically been more than 1 FLOPS/Byte and expected to increase even more in the future [111]. Therefore, the program still remains memory bandwidth bound according to the roofline model for performance of multicores [140]. Section 3.4 shows how our approach overcomes this bottleneck by incorporating *time-tiling* [53] to increase the data reuse from the caches.

3.3 Execution of dynamic task graphs

In the previous section we saw how over-synchronization effects the scalability of stencil computations. In order to reduce this synchronization overhead we program stencil operations as dynamic task graphs (section 2.1.1). These dynamic task graphs are executed with a dynamic work-stealing scheduler. However, the seemingly simple execution semantics of dataflow are not trivial to maintain in dynamic task graphs, where

nodes and edges are added and removed at runtime. Two hazards that may result in deadlocks or data races are: (1) *dead predecessors* and (2) *unborn predecessors*. We use a dynamic task graph model of a hypothetical application in figure 3.3 to explain these hazards.

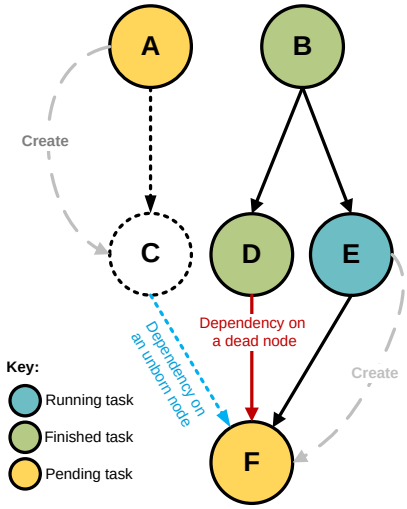


Figure 3.3: A dynamic task graph snapshot highlighting the hazards of dynamic node creation. Task E is executing and creates task F, tasks B and D are already finished, whereas task A will create task C when it executes sometime in the future

1. *Dead predecessors:* In a dynamic task graph, tasks execute only once over the span of the application. A node may be created after one or more of its predecessors have finished execution (*dead nodes*). For example the task D in figure 3.3 has finished execution at the time when the task F is created. Therefore task D will no longer send out any more messages. Task F will keep waiting for input from task D forever, and will never execute. In order to avoid this deadlock, an edge must never be created from a dead node.

2. *Unborn predecessors:* In dynamic task graphs, the programmer does not have any control over the sequence in which the nodes are scheduled and it is possible to create a task before its predecessor is

created. However, it is not possible to create an edge from such an *unborn* task. For example, the task F in figure 3.3 is created before its predecessor task C, which will only be created when task A executes. If this dependency is completely omitted F may execute before C is finished, causing a data race.

Even though generic solutions to the above mentioned hazards do not exist it is possible to deal with them in application specific manner. In section 3.4 we show how some properties of stencil computations may be used to avoid the above mentioned hazards. The remainder of this section explains how synchronization overhead is reduced by executing task graphs with a work stealing scheduler and why garbage collection is difficult in dynamic task graphs.

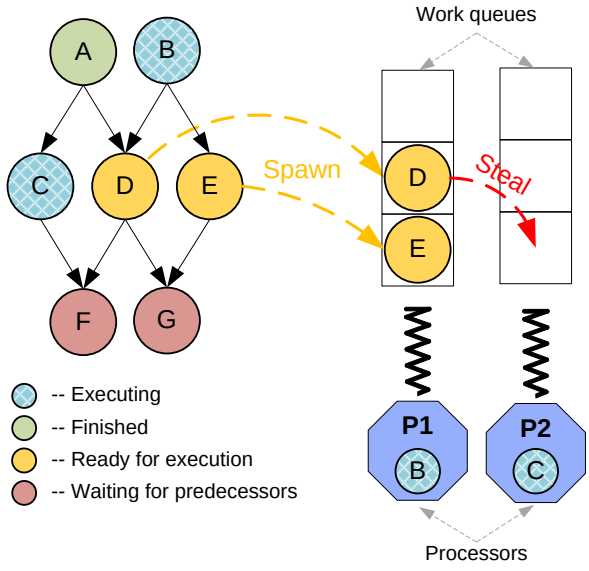


Figure 3.4: A scheduling illustration showing a snapshot of the DTG when the tasks C and B finish execution on the processors *P1* and *P2* respectively.

3.3.1 Minimizing synchronization overhead

Work-stealing schedulers are a proven way to execute task graphs [108]. In a work stealing scheduler, each processor runs a work-stealing thread (worker). Each thread has a pool (queue) of tasks waiting to be executed called a *work queue*. The worker threads execute tasks from their own work queues as long as they are not empty. When the work queue becomes empty the worker tries to randomly steal a task from the pools of other workers. A task becomes ready for execution after it has received messages on all its incoming edges. When a task becomes ready for execution it is spawned as a task in a work queue. Thus, the scheduling is a combination of work stealing and dataflow.

Figure 3.4 shows a snapshot to illustrate the scheduling mechanism of dynamic task graphs. When the task B finishes execution messages are sent to the tasks D and E which are then ready for execution and are therefore spawned on the work queue of processor *P1*. When task C completes a message is sent to the task F. However, it can not be spawned on a work queue until it receives a message from the task D as well. The work queue of *P2* then becomes empty. Therefore, it steals a task from the work queue of *P1*.

Note that tasks are only spawned on a work queue when all the data required for their execution is available and the cores executing these tasks do not have to synchronize during their execution, thus further reducing synchronization overhead. Our results also show that all cores are fully utilized and lazy task spawning does not decrease processor utilization for stencil computations.

3.3.2 Memory de-allocation

In applications based on iterative algorithms, the number of tasks created at runtime may be large (in millions). For such an application, a dynamic task-graph based system might run out of memory in the absence of a task de-allocation scheme. Since DTGs allow arbitrary dependencies between any pair of tasks in the dynamic graph, the results computed by a task may be required not only by other tasks that exist at present but also by tasks yet to be created in future.

Due to the difficulties in analyzing the dataflow of future tasks a generic memory de-allocation scheme that can remove finished tasks and their results, does not exist [4]. However, for applications with more regular dependency patterns, such as for stencil computations, it is possible to devise application-specific memory de-allocation schemes. This is outlined in the following section.

3.4 Stencil computations as dynamic task graphs

In this section we describe how stencil operations are expressed as dynamic tasks graphs. Figure 3.5 shows a dynamic task graph for a 2-dimensional 5 point stencil computation. The domain is divided into four tiles A, B, C and D and initial tasks which perform the stencil operation on these tiles are created. These initial tasks then create successor tasks that work on the same regions for the subsequent timesteps. The left hand side of figure 3.5 shows the upper right corner of tile B being updated, the data required for this operation resides not only in the tile B itself but also in the tiles A and D. Therefore, a task updating tile B in the timestep t may not start before the tasks working on the tiles A, B, and D in timestep t-1 are finished, the DTG takes all these dependencies into account. Note that these dependencies cannot be expressed using spawn-sync or fork-join mechanisms without using all-to-all barriers. This is why our dynamic task-graph based stencil operations minimize synchronization costs, which will be shown in Section 3.6.

3.4.1 Avoiding dynamic task creation hazards: dead and unborn predecessors

During the initialization phase tasks are created for the first two timesteps to solve regions of the matrix e.g. tiles for a 2D domain. These initial tasks then create other tasks (along with their incoming edges) working on the same regions for the subsequent timesteps. Tasks are created two timesteps in advance e.g. the task A_0 in figure 3.5 creates the task A_2. This ensures that deadlocks due to *dead predecessors* do not occur.

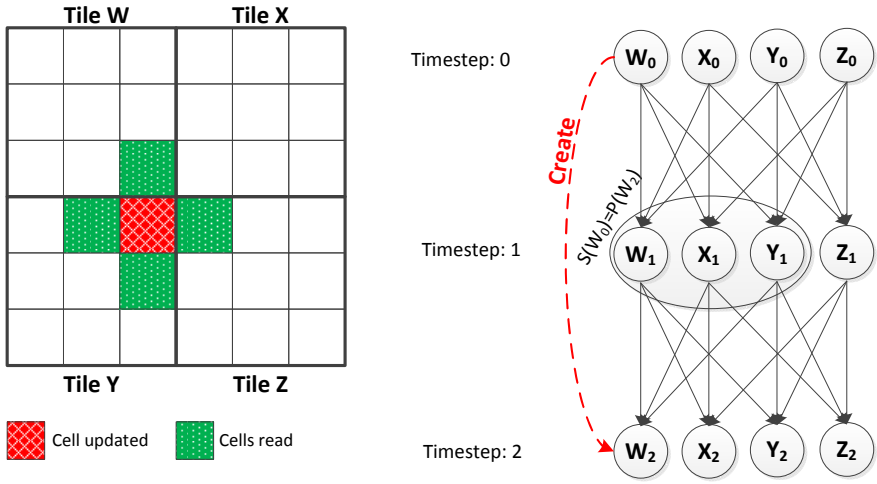


Figure 3.5: (a) Shows a DTG for a square tilled 2-dimensional 5 point stencil computation, the green regions show the data dependencies for updating the red regions. (b) Shows a corresponding dynamic task graph A_0, B_0, C_0 and D_0 are tasks that perform the stencil operation on the tiles A, B, C and D in timestep 0.

Let A_n be the task that processes the region A in the n^{th} timestep. Let $P(A_n)$ and $S(A_n)$ be the sets of the immediate predecessors and successors of A_n respectively. Also let $CreationTime(A_n)$, $StartTime(A_n)$ and $FinishTime(A_n)$ be the creation, starting and finishing times of the task A_n. For any stencil of a constant shape and order $P(A_{n+2}) \equiv S(A_n)$, i.e. all immediate predecessors of A_{n+2} are the immediate successors of A_n e.g. the predecessors of task A_2 in figure 3.5 are A_1, B_1 and C_1, all of whom are successors of A_0. If the task A_n creates the task A_{n+2} sometime during its execution, $CreationTime(A_{n+2}) < FinishTime(A_n)$. Since a task may not start before all its predecessors are finished, $\forall_{Task \in S(A_n)}, StartTime(Task) > FinishTime(A_n)$. Therefore, $\forall_{Task \in P(A_{n+2})}, StartTime(Task) > CreationTime(A_{n+2})$, i.e. none of the predecessors of a task may start before it is created.

Since all tasks are created after their predecessors the possibility of *unborn predecessors* does not exist in regular stencil computations.

3.4.2 Memory de-allocation

Garbage collection schemes for generic DTGs do not exist, due to the difficulties of analyzing data dependencies of future tasks. Many applications that use stencil computations typically have a large number of timesteps (in millions at least). Therefore it is not feasible to use dynamic task graphs for stencil computations without garbage collection or a memory de-allocation scheme. We use properties of the dependency structure of stencil computation to devise a memory de-allocation scheme for stencil computations.

By definition all tasks that requires the results of the task A_n are included in the set $S(A_n)$. For a regular stencil $S(A_n) \equiv P(A_{n+2})$. Therefore, A_{n+2} may only start when all the tasks in $S(A_n)$ are finished. Thus it is safe to delete or overwrite the results of A_n once the execution of A_{n+2} starts, as illustrated in figure 3.5.

3.4.3 Time tiled stencil computations

Time tiling is a technique used to increase the computation intensity of the stencil operation in order to circumvent the memory bandwidth bottleneck. The idea is to subdivide the original space-time domain in smaller sub-domains with a minimal surface to volume ratio and with a volume that fits in the faster cache memory. This improves data locality, decreases the number of cache misses and generally improves performance [15, 37, 53, 118, 126].

For stencils, tiling within a single iteration (one step of the `t` loop) only leads to minimal improvements in data locality. However, by tiling in both the simulation space (`x` and `y` loops) and time (`t` loop), data locality can be improved drastically. Since the stencil prescribes a certain dependency between a point at one time step and points on a previous time step, the shape of the tile cannot be chosen arbitrarily. We have chosen to use *square frustum* shaped tiles, as shown in figure 3.6. This square frustum, a pyramid without its top, has its bottom plane in the `x`-`y` domain and extends upwards in the `t` direction. Since for the five-point stencil, only two grids have to be kept in memory, the size of the frustum must be chosen such that two of its bottom planes fit in cache

memory at the same time. However, to iterate over all grid points, in all time steps, an overlap between the different tiles is required. Figure 3.6 illustrates the shape of the tiles with their overlap for a two-dimensional five-point stencil.

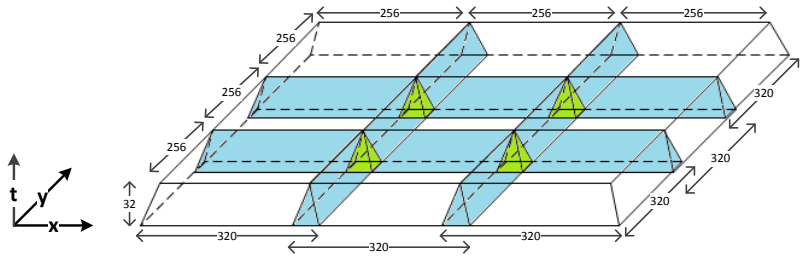


Figure 3.6: Frustum based time tiling of a 2D stencil operation. Horizontal and vertical overlapping regions are shaded blue, whereas, diagonal overlapping regions are shaded green.

Wherever the tiles overlap, computations have to be performed redundantly. Although increasing the height of the frustum leads to better data reuse by doing more work on data already in fast memory, it also increases the amount of redundant computations. The optimal tile height will depend on this trade-off. In our experiments, we have taken tiles with a $(256 + 2 \times 32) \times (256 + 2 \times 32)$ base and height 32. The arithmetic intensity can be computed as follows. For each tile, one 320×320 grid has to be loaded from and one 256×256 grid written to main memory. The total number of flops per tile is the number of grid points per tile[2], $(160 \times 320^2 - 128 \times 256^2)/3$, times 7 FLOPS per grid point. This results in an overall arithmetic intensity of 13.9 FLOPS/byte, which definitely makes the computation compute bound rather than memory bandwidth bound. As the numerical results section will show, the benefit of improved data locality outweighs the redundant computations.

This tiling approach also reduces synchronization overhead, since synchronization only occurs after multiple timesteps. However, our

[2]Volume of a frustum is volume of a pyramid minus its top. Volume of a pyramid is $\frac{1}{3}Bh$ with B the area of the ground surface and h the height.

results (section 3.7.1) show that the all-to-all barriers are still expensive and their overhead is severely aggravated with load balancing problems. Therefore, we consider each tile as a task in the dynamic task graph approach and take into account the dependencies between the tiles. This avoids all explicit barriers.

Data dependencies of time tiled stencil computations The dependencies of a simple tiled (non time-tiled) version as shown in figure 3.5 are similar to the stencil operation itself, where every tile requires values from its horizontal and vertical neighboring tiles[3]. However, when time-tiling is applied the resulting frustums not only require values from their horizontal and vertical neighbors but also from their diagonal neighbors. This is illustrated in figure 3.6. The horizontal and vertical overlapping regions are shaded blue whereas the diagonal overlapping regions are shaded green. Also note that the horizontal and vertical overlapping regions have a computational redundancy of 2, whereas, the diagonal overlapping regions have a computational redundancy of 4.

3.5 Implementation

In order to validate and benchmark our approach we first need to implement it. The implementation uses TBB flowgraphs as an enabling technology to implement dynamic task graphs and execute them with the TBB scheduler. As alternative for TBB flowgraphs we could have used the Nabbit library [4], which is conceptually similar but unfortunately is no longer maintained and relies on a now-obsolete version of Cilk. In the remainder of this section we first give a brief introduction to TBB Flowgraphs and then explain the mapping of DTGs to TBB Flowgraphs.

3.5.1 TBB Flowgraphs

The Intel Threading Building Blocks (TBB) Flowgraph [77] provide abstractions that allow the programmer to express algorithms in the

[3]Considering x-axis and y-axis as the horizontal and vertical domains respectively.

form of graphs that represent the data and control flow of the program. A flowgraph consists of two types of objects: 'nodes' and 'edges'. The nodes are further classified into four categories:

1. Functional: source, continue node, function node and multi-output function nodes.
2. Buffering: buffer node, queue, priority queue & sequencer nodes.
3. Split/Join: queuing join, reserving join, tag matching join & split nodes.
4. Others: broadcast, write once, overwrite & limiter nodes.

Detailed documentation on the syntax and semantics of all these nodes is included in the TBB reference manual and beyond the scope of this thesis. We focus our discussion on the function nodes and queuing join nodes. A *function node* executes a block of code whenever it receives an input on one if its incoming edges and broadcasts the output on all its outgoing edges upon completion. A *join node* gathers inputs from all its incoming edges and forwards them in the form of an output tuple containing one message from every input.

Consider the flow graph example shown in figure 3.7. The graph g consists of five *function nodes* a,b,c,d,e and one *join node*. The *function nodes a, b, c, d, e* execute the user provided functions *F(), G(), H(), I(), J()* respectively. The execution of the graph starts when an input message is sent to the node a using the $try\_put()$ function. The node a executes the function *F()* and sends a message to the nodes b and c. The node d can start execution as soon as it receives a message from either node b or node c i.e. as soon as one of its predecessors finishes execution. The *join* node waits for one input from both nodes b and c before sending them as a tuple to node e, therefore, the node e can only start after both its predecessors b and c are finished. Also note that node d fires twice during the execution of the graph (once for every one of its predecessors), whereas node e only fires once. The $g.wait\_for\_all()$ function returns when there are no more messages to process in the graph 'g' and none of the nodes are executing.

```
using namespace tbb::flow;
int main() {
    graph g;
    //creation of graph nodes
    function_node<input,ouput> a( g, F());
    function_node<input,ouput> b( g, G());
    function_node<input,ouput> c( g, H());
    function_node<input,ouput> d( g, I());
    join_node<input_tuple> join;
    function_node<input_tuple,ouput> e( g, J());
    //creation of edges between the nodes
    make_edge(a, b);
    make_edge(a, c);
    make_edge(b, d);
    make_edge(c, d);
    make_edge(b, join);
    make_edge(c, join);
    make_edge(join,e);
    //start execution
    a.try_put(input);
    //trigger the execution of the graph
    g.wait_for_all();
    //wait till no more messages in the network
    return 0;
}
```

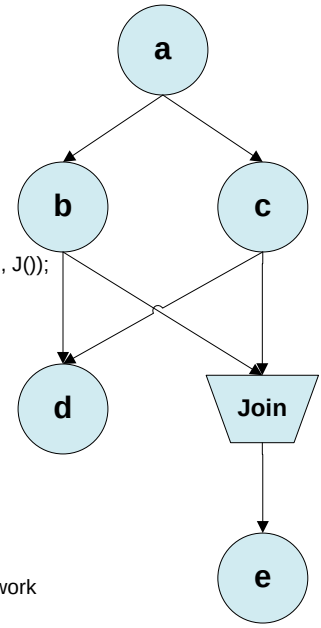


Figure 3.7: Example of a static dependence graph using TBB Flowgraphs.

3.5.2 Dynamic task graphs using TBB Flowgraphs

The *function nodes* and *edges* in flowgraphs are used to model the *tasks* and *edges* of dynamic task graphs. However there is a subtle difference in the execution semantics of the two models. DTG tasks only fire after receiving a message from all predecessors, whereas a flowgraph function node fires as soon as it receives a message at one of its inputs. We implement DTG tasks using flowgraph constructs as shown in figure 3.8. Placing a join node preceding a function node ensures that it is only executed after all the predecessors have finished, thus preserving the execution semantics of task graphs. DTG edges are similar to flowgraph edges, however, the *make_edge* function needs to be modified as shown in figure 3.8. An edge between a pair of DTG tasks is created by making a flowgraph edge from the function node of the source task to the join node of the destination task.

```
using namespace tbb::flow;
class DTG_task {
        function_node<in_tuple,out_tuple>* func;
        join_node<in_tuple>* join;
        DTG_task (int A,int B)  {
                node = new function_node<in_tuple,out_tuple>
                (g,1,[=](const in_tuple &message)
                ->in_tuple{return function(A,B,message);});
                join = new join_node<out_tuple>(g);
                make_edge(*join,*node);
        }
        ~DTG_task () {
                delete func;
                delete join;
        }
};
void DTG_make_edge(DTG_task* source, DTG_task* destination) {
        make_edge(source->func, destination->join);
}
```

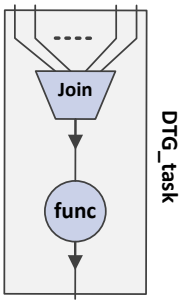


Figure 3.8: Mapping DTG tasks to TBB flowgraphs

3.5.3 Optimizing runtime edge creation for stencil computations

Creating edges at runtime requires searching the graph structure to find predecessors. For a stencil computation with large domains this graph structure may be large, causing a substantial overhead for edge creation. Therefore, we organize the dynamic task graph in the form of linked lists as shown in figure 3.9. Tasks that work on the same region are connected via a linked list and tasks within the linked list are identified by the timesteps they compute. A directory (array of pointers) contains the addresses of the heads of all linked lists. If a task needs to find another task e.g. for edge creation, it first consults the directory for the head of the linked list using the spatial coordinates of the required task and then searches the linked list for the required timestep.

3.6 Performance evaluation

For evaluation we implemented two benchmark applications in with our approach (Parody): (1) Conway's Game of Life [55] and (2) Jacobi

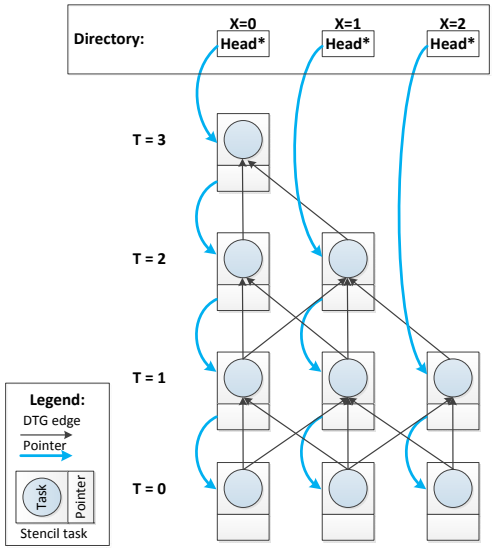


Figure 3.9: A one dimensional 3-point dynamic task graph organized as linked lists for faster edge creation

solver for the heat equation. We selected these applications because their implementations for the state-of-the-art stencil compilers Pochoir [126] and Pluto (with diamond tiling) [15] are available, making a direct performance comparison possible. The performance benchmarking experiments are performed on a dual socket Xeon® E5-2670 system with a total of 16 cores.

3.6.1 Game of life

The game of life benchmark employs a 9-point stencil operation on a two dimensional Boolean grid of size 8192×8192 and runs for 2048 timesteps. Figure 3.10 shows the execution time for three implementations of this application i.e. Pochoir, Pluto and DTG. The results show that Parody outperforms the other approaches in this experiment. With 16 cores Parody performs at 7.23 billion lattice updates per second whereas Pluto

and Pochoir versions produce at 4.89 and 3.46 billion lattice updates per second respectively.

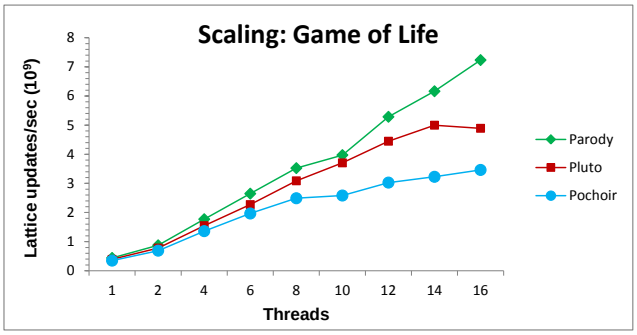


Figure 3.10: Strong scaling results of Conways game of life benchmark on dual socket Xeon® E5-2670, domain size 8192×8192 (higher is better).

Besides these scaling experiments on the 16 core machine we also performed a larger scale experiment with a domain size of 16384×16384 on a 120 core shared memory system. The Pluto and Parody version clocked at 12.54 and 37.72 billion lattice updates per second, resulting in net performance improvements by a factor of 3.01.

3.6.2 Heat benchmark

The heat benchmark employs a 5-point stencil operation on a two dimensional double precision grid of size 8192×8192 and runs for 2048 timesteps. Each stencil update requires seven floating point operations. Figure 3.11 shows the average speed in billion lattice updates per second, the values reported are for the useful work done only i.e. not counting in the redundant work done due to time tiling. Again the results show that Parody outperforms the other approaches. With 16 cores Parody performs at 9.88 billion lattice updates per second whereas Pluto and Pochoir versions run at 5.61 and 8.28 billion lattice updates per second respectively. The performance degradation for Pluto at 14 and 16 threads is because of workload balancing problems.

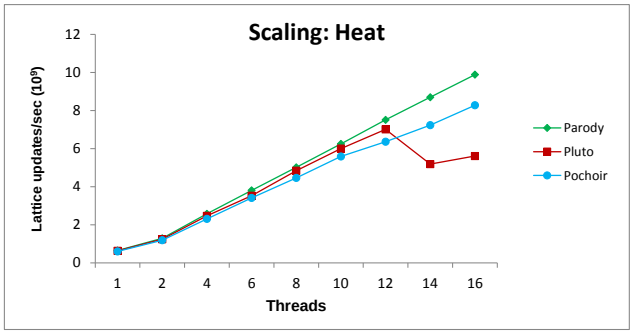


Figure 3.11: Strong scaling results of the 2D heat simulation benchmark on dual socket Xeon® E5-2670, domain size 8192×8192 (higher is better).

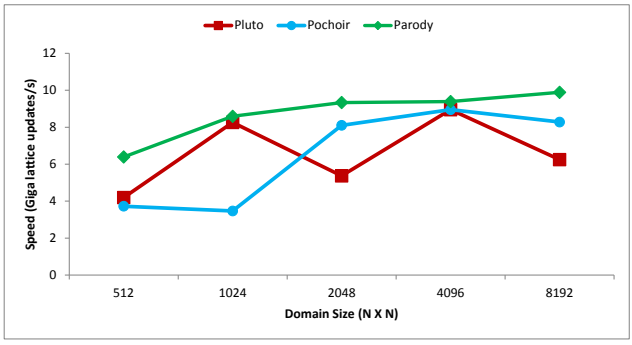


Figure 3.12: Performance vs domain size for the 2D heat simulation benchmark on dual socket Xeon® E5-2670 (higher is better)

Figure 3.12 show the performance of the three implementations with respect to increasing domain size. Parody outperforms the other two implementations by varying margins. For small domain size (i.e. 512×512) the synchronization overhead dominates the cost for both the pochoir and pluto implementations, thus, we see a huge gap in performance. The workload imbalance for the pluto implementation varies with the domain sizes, thus, it causes a varying amount of synchronization overhead. The pochoir implementation has a generally better workload distribution and shows a more consistent performance for higher domain sizes. However, the time-tiling strategy of Pluto is more advanced. Therefore, it produces

good results if the workload is well balanced.

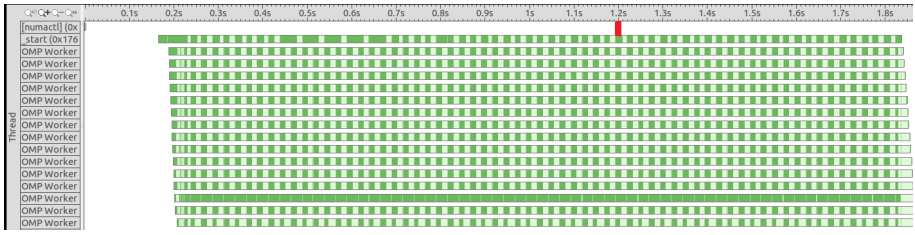
3.7 Discussion

In the previous section we saw that the Parody approach outperforms the state-of-the-art stencil compilers, in this section we give an in depth discussion on why this approach is faster. First we compare the synchronization overhead of our approach with the state-of-the-art, followed by some discussion on vectorization speed-up and then we evaluate the impact of an aggressive NUMA optimization applied to our approach.

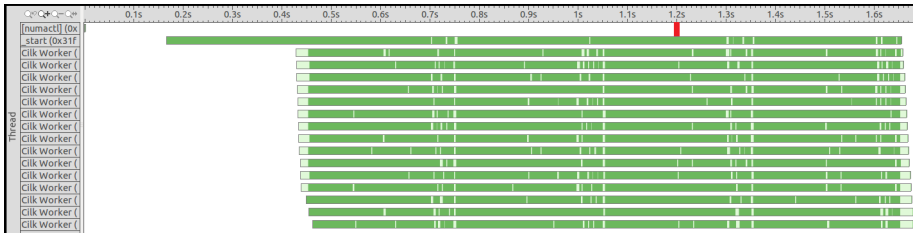
3.7.1 Synchronization overhead

We use the Intel® VTune™ profiler locks-and-waits analysis to measure the synchronization overhead. Figure 3.13, shows the runtime synchronization behavior of the Pluto, Pochoir and Parody versions, for a smaller version of the heat benchmark with the domain size of 2048×2048 on a dual socket Xeon® E5-2670. Each green bar shows the execution of a thread with time on the horizontal axis. The dark green portion represents the useful work, whereas the light green portion represents synchronization overhead.

Figure 3.13(a) shows the execution of the Pluto implementation, based on OpenMP. This implementation uses diamond shaped time-tiles to overcome the memory bandwidth limitation and uses a *"parallel for"* loop to compute them. An implicit barrier is encountered in each iteration of this loop, as clearly visible in the figure. Even though time-tiling reduces synchronization to some extent, 46% of the total CPU time (for all threads) is lost due to a combination of load balancing and over synchronization problems. We also see from the same figure that one of the OpenMP workers is allocated more work compared to the others and all the other workers have to wait for it at the barrier.



(a) Pluto



(b) Pochoir

(c) Parody

Figure 3.13: Intel® VTune™ Amplifier screen-shots of the locks-and-waits analysis, for the 2048×2048 grid experiments of the heat benchmark on dual socket Xeon® E5-2670. Each green bar shows the execution of a thread with time on the horizontal axis. The dark green portion represents the useful work, whereas the light green portion represents synchronization overhead. Note that the timescales on the three graphs are different, observe the red markers that indicate 1.2s of execution time.

Figure 3.13(b) shows the execution of the Pochoir implementation. This implementation uses space-time hyper-trapezoidal time-tiling and an Intel Cilk based spawn-sync structure to compute these hyper-

trapezoids. From the figure we can see that it suffers less synchronization problems and better workload distribution as compared to the Pluto implementation. However, a substantial 20% of the total CPU time is still consumed in synchronization.

Figure 3.13(c) shows the execution of the dynamic task graph version, where less than 2% of the CPU time is spend on synchronization during the execution of the stencil kernel.

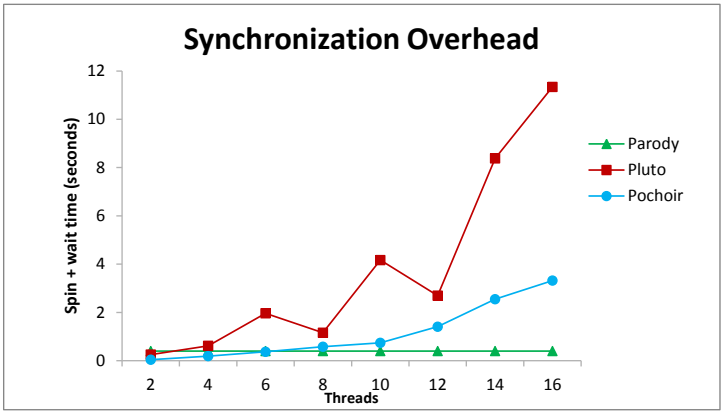


Figure 3.14: Synchronization overhead (less is better) for the 2D heat benchmark with respect to threads, the domain size is 8192×8192 and a dual socket Xeon Sandy Bridge E7-2670.

Figure 3.14 shows the synchronization costs with respect to the increasing number of threads. Apart from the fact that the synchronization costs of the DTG implementation at 16 threads is at least ten times less than the other implementations, there are two important observation to be made from this experiment. Firstly the synchronization cost grows greatly with the number of threads for both the approaches that use barrier synchronizations. This is due to two reasons: (1) there are more and more threads waiting on a given barrier and (2) the barriers become more costly as the number of threads increase. Secondly load balancing problems may increase the synchronization costs by several folds, as in the case of the Pluto implementation. Figure 3.15 shows the synchronization overhead

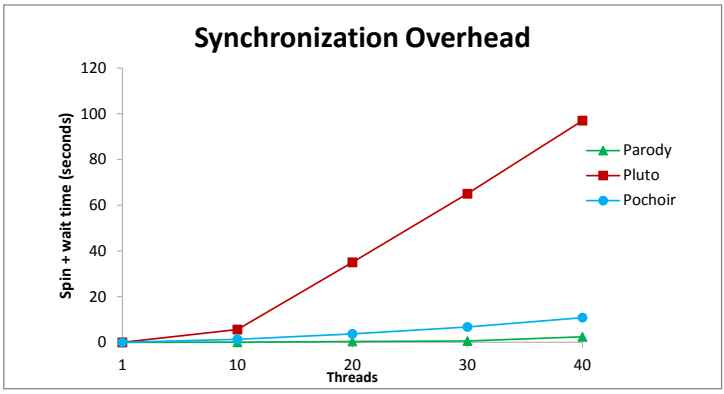


Figure 3.15: Synchronization overhead on a 40 core system (4 socket, 10 Westmere-EX E7-4870).

on a 40 core system[4]. At 40 threads this shows very low synchronization overhead for parody (less than 1 seconds), a larger overhead for Pochoir (8 seconds approx.) and a huge overhead for Pluto (over 100 seconds).

3.7.2 Vectorization

Figure 3.16 shows a comparison of the single core vectorization speed-ups for the three implementations of the heat benchmark on two different CPUs; a Westmere X5660 with 128-bit wide vector instructions and a Sandy Bridge E7-2670 with 256-bit wide vector instructions. Only compiler generated vectorization was used for all three implementations, on both platforms. We observe a substantial difference in the speed-ups on this platform, even though the same stencil operation is being vectorized. One reason for this difference are the shapes used for time-tiling the stencil operation[5]. Pluto uses tetrahedrons and pochoir uses hyper-trapezoids; these shapes sometimes result in narrow loop bounds for the innermost loops (e.g. in the apex of tetrahedrons for Pluto).

[4]Note that the results presented in figures 3.14 and 3.15 involve two different types of processors i.e. Xeon Sandy Bridge for figure 3.14 and Xeon Westmere EX for figure 3.15.

[5]Differences in memory alignment for example due to zero padding in the pochoir implementation also affect vectorization speed-ups.

Our DTG implementation uses frustums for time tiling thus always has wide loop bounds for the two innermost loops, resulting in better vectorization[6]. However the price is paid in the form of some overlapping regions resulting in some redundant work, for our approach. We also observe that different approaches scale differently with the increase in the width of vector instructions. These tradeoffs will become more important with the increasing vectorization widths in future CPUs [111] and further research is required to find the optimal domain decomposition and time tiling strategies that take vectorization widths into account.

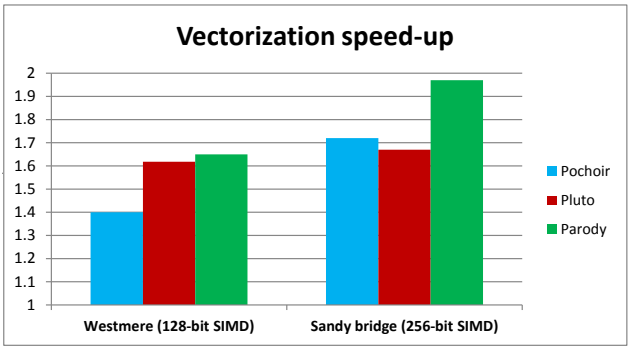


Figure 3.16: Single core vectorization speed-ups for the heat benchmark

3.7.3 Cost of re-establishing NUMA locality at runtime

In systems with logically shared but physically distributed memory it is generally important to take Non-Uniform Memory Access (NUMA) effects into account. On a multi-socket system, for example, a CPU has access to the complete RAM memory of the system but with different latencies. Even though TBB's scheduler has a weak data locality awareness in its "Breadth-First Theft and Depth-First Work" strategy [108], it tries to maintain maximum processor utilization through aggressive work-stealing. Tasks are occasionally stolen by cores on a different socket resulting in the loss of NUMA locality. The price of re-establishing this locality is coping data along with the tasks.

[6]We do not claim that a frustum is the most optimal shape for time-tiling stencil operations.

In order to evaluate the feasibility of aggressively maintaining NUMA locality at run time, through the life span of the application[7], we implemented a NUMA aware version of the heat benchmark using the *libnuma* API. Every task communicates its NUMA domain to its *direct successor*[8]. When a successor finds its own NUMA domain different from its predecessor it copies the data to its local memory, thus re-establishing NUMA locality.

For a domain size of 8192×8192 the overhead of NUMA awareness is 2.5% of the execution time. However, it did not yield any performance benefit for this benchmark, because of two reasons: (1) the algorithm is not bandwidth constrained due to time-tiling, (2) compiler based prefetching hides the latency of accessing the remote memory. Nonetheless we expect this tradeoff to be significantly different for higher order stencil operations, where memory bandwidth saturation problems are more severe. Further investigations are required to evaluate the feasibility of such aggressive runtime NUMA optimizations for higher order stencil operations.

3.8 Related work

Stencil operations are a very frequently recurring design pattern, hence they are very extensively studied as well. With the recent advent of multi-core processors many stencil compilers and auto-tuning frameworks have spurred. Most of these stencil compilers and auto-tuning frameworks focus on increasing the arithmetic intensity of the kernel, hence reducing cache misses in order to bypass the memory bandwidth saturation and reduce synchronization. These tools usually require the stencil operation to be expressed in *domain specific language* (DSL) and generate the optimized implementation in C, FORTRAN or CUDA etc.

The Pochoir stencil compiler [126] cuts the domain into space-time hyper-trapezoids using the concepts formalized by Frigo and Strumpen in [53]. These zoids are then classified into groups, such that all zoids in one

[7] Note that initial NUMA-aware allocation is used in all cases.

[8] A *direct successor* is a task that works on the same region but for subsequent timesteps.

group may be solved in parallel, i.e. precedence constraints are only between the groups and not within a group. The shapes of these zoids allow calculation of multiple timesteps without synchronization because all the data required to calculate the next timesteps is either within the zoid or in a predecessor zoid. Hence, this domain decomposition improves data locality and reduces synchronization overhead. Similarly [15] presents a stencil version of the Pluto compiler [19], that uses "diamond" (tetrahedron) shaped time-tiles for the domain decomposition of stencil operations. However, in contrast to our approach both these stencil compilers use a *"parallel for"* loops to realize the parallelization, thus placing an implicit synchronization barrier between subsequent time steps. Other approaches that time-tile across several timesteps include [38, 44, 63, 137]. In this chapter, we took Pochoir and Pluto as a representative of the state-of-the-art for comparison with our approach. We showed how both were outperformed because the synchronization costs become ever more expensive. In other words: the synchronization costs we eliminate were holding back the algorithmic improvements advocated by these papers.

In auto-tuning frameworks such as Patus, the user may also provide a parametrized parallelization strategy and an XML description of the platform, along with the specification of the stencil expressed using a domain specific language. Some basic strategies (e.g. basic outer loop parallelization and cache blocking), along with XML descriptions of some common platforms (e.g. Intel x86_64 and NVIDIA Tesla) are already available in the framework. The framework then generates (OpenMP [42] based) parallel code for the kernel and an auto-tuning script for exploring the parameters for the strategy. This script executes the kernel repeatedly with different parameters to find the best configuration. Unlike our approach that uses point-to-point synchronization Patus uses explicit OpenMP barriers between timesteps. Other frameworks in similar spirit include [74] and [131]. Our experiments clearly showed that this becomes very costly as the number of threads increases.

Phasers [113] are point-to-point synchronization mechanisms for shared memory platforms. A thread/task can register on a set of Phasers and synchronize with other threads/tasks using signal-wait mechanisms. In the Phasers approach tasks/threads in the wait stage are blocked until

they receive signals from all their registered Phasers, whereas in our approach approach task assignment to worker threads is delayed for synchronization, therefore, the CPUs are free to execute other tasks that are ready for execution. A stencil implementation that uses Phasers for synchronization instead of barriers is presented in [112]. Unfortunately the stencil algorithm used in their approach does not apply time-tiling, therefore, a fair head-to-head comparison with our approach is not possible. A technique using a combination of low level locks and barriers is proposed in [118], however, unlike our approach or Phasers this is not a complete point-to-point synchronization; i.e. group synchronization mechanism are still used.

A static task-graph implementation of a stencil operation is presented in [105]. Schedules are compiled off-line before the application starts. At runtime the tasks are executed in predefined time-slots on the different processors. When task graphs are scheduled statically no synchronization is required at runtime. However, this approach is not able to cope with dynamism whether it is from the platform itself i.e. in the from of dynamic voltage and frequency scaling or from other applications/processes sharing the system.

Nabbit [4] is a library for expressing dynamic task graphs which can be executed using Cilk. The nabbit interface does not allow garbage collection of nodes at runtime. Therefore it is not possible to directly use it for stencil operations with large number of timesteps, as the system runs out of memory quickly.

3.9 Conclusion

This chapter proposes to implement stencil computations as dynamic task graphs in order to minimize synchronization overhead. The approach we propose avoids global barriers, synchronization is only point-to-point and because tasks are only created when they are ready to be executed there is minimal spin-waiting i.e. non-blocking synchronization. We use a frustum shaped time tiling for improving data locality and thus reducing cache misses.

Our results show that approach scales better compared to two state-of-the-art stencil compilers Pochoir [126] and Pluto [15] for the two benchmarks used in our experiments: (1) Conway's game of life and (2) 2D heat simulation. Detailed profiling shows that the synchronization overhead in our approach is at least 5 times less than Pochoir and 50 times less than Pluto. Analyzing vectorization speed-ups shows that our frustum shaped time tiling vectorized better than the two stencil compilers and that this difference becomes more significant with the increasing number of SIMD lines.

The next chapter studies improvements in the mapping and scheduling of embedded applications on MPSoCs.

Chapter 4

Mapping applications to MPSoCs

The previous chapter used dataflow models to optimize a kernel in the domain of high performance computing. Another big driver for parallel programming is embedded software. Synchronous dataflow graphs [85] and it variants (see the taxonomy in figure 2.2.2) are often used to represent embedded streaming applications. This chapter explores Pareto optimal mapping of such applications in the context of quasi-static scheduling (see section 2.2.3). Two application use cases are used for evaluation; an image processing application and an H.264 video decoder.

4.1 Introduction

Nomadic devices such as smartphones and tablets have become omnipresent. We want to surf the web, watch videos and play games on these devices anytime anywhere. In order to give the users a seamless experience these devices have to process large amounts of data in strict timing deadlines, which drains a lot of power. Therefore, battery time (time between successive chargings) remains a big concern.

Modern nomadic devices are often based on heterogeneous MPSoC platforms, with complex memory hierarchies and interconnects. Figure

4.1 shows a skeleton of a bus based heterogeneous MPSoC with a two level memory hierarchy and four cores of two different types. The low power cores are often simpler and smaller RISC processors. The high performance cores are usually larger DSPs, VLIW processor, SIMD processors or GPUs. All these processor have different power modes that need to be configured at runtime. The different types of memories provide different data access latencies and bandwidths. The caches and scratchpads provide lower latency, higher bandwidth and are more energy efficient than the main memory. However, they are small and private, whereas, the main memory is larger and shared. The communication bus often becomes an important bottleneck as the processors begin contending for the it.

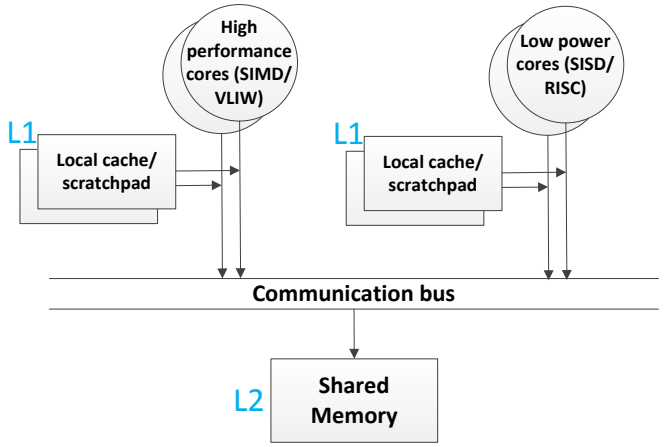


Figure 4.1: Anatomy of an MPSoC platform

Mapping a software application on an MPSoC platform requires scheduling of three different types of *activities* onto three different types of *resources*:

1. Computational tasks on the different cores
2. Data storage in the memories
3. The data transfers on the interconnect

The increasing complexity of the hardware platforms makes it very difficult for the software developers to do this mapping manually. Embedded applications have also become very dynamic, making it impossible for compilers to statically make all the scheduling and resource management decisions, at design time. Embedded and real-time operating systems cannot manage this orchestration in a fine grained manner at runtime because of the prohibitive overhead of computing these resource management and scheduling decisions.

To reduce this overhead researchers have studied two-phased mapping techniques [61, 117, 120] for configuring the hardware and scheduling the software. Two-phase mapping techniques split the mapping into design-time and runtime phases. At design-time, Pareto-optimal mapping solutions [1] are computed off-line for several possible runtime scenarios [61]. At runtime, these precomputed schedules are dynamically activated depending on the runtime situations. These approaches have shown to be capable of handling different types of dynamism with low overhead [61, 117, 120]. This chapter focuses on the memory and communication-aware design-time phase of mapping applications to MPSoCs.

The problem of design-exploration of mapping solutions can be split into two parts: (1) Spatial assignment and (2) temporal scheduling. *Spatial assignment* decides where an activity takes place without specifying when. For example, only deciding which processor a task executes on or which memory a data object is stored in, without specifying a time-slot. *Temporal scheduling* allocates time-slots for these activities on the specified resources.

In this chapter we present a novel approach to the formulation of the memory and communication-aware mapping problem that allows an exploration of spatio-temporal schedules for computation, data storage and communication, in a tightly coupled fashion. Two aspects of this coupling are explored:

[1] A set of solutions such that every solution is better than all other solutions w.r.t at least one criterion

1. *Horizontal coupling:* is the coupling between different domains (i.e. memory, communication and processing)

2. *Vertical coupling:* is the coupling within the same domain (e.g. scratchpad allocation aware buffer dimensioning or task allocation aware selection of CPU power modes)

The next section describes these horizontal and vertical coupling aspects of the mapping problem in detail.

Several memory and communication-aware design-time mapping exploration techniques have been proposed [24, 50, 51, 64, 89]. The techniques presented in [64, 89] split the mapping problem into two sub-problems of spatial assignment and temporal scheduling and solve them separately. This reduces the search space at the cost of the quality of the solution. The approach proposed in this chapter will instead focus on higher quality solutions (i.e. better energy efficiency or throughput of the mapping solutions) without exorbitant computational complexity. The approach presented in [51] uses the ant colony optimization in a multi-stage implementation, but unlike our approach it only optimizes the execution time of the application ignoring the energy consumption. An other approach [50], optimizes the energy consumption along with the execution time of the application but defines the mapping only as a spatial assignment problem, without the temporal aspect of scheduling, which we take into account.

We validated our proposed technique by using it to find the energy-speed trade-offs for two benchmark applications: (1) a cavity detector application [26] and (2) an H.264 decoder. The target MPSoC consists of four Strong ARM 1100x processors and two TI-C64X+ processors connected via a shared bus (see figure 4.1). In our experiments, we observed that tightly coupling mapping solutions are significantly better when compared to decoupled exploration, for both benchmarks. The Pareto-optimal mapping solutions found by our exploration technique can be used by the runtime managers of the two-phased mapping techniques [61, 117, 120].

The rest of the chapter is structured as follows. Section 4.2 describes the horizontal and vertical coupling. Section 4.3 gives an overview

of the design-time co-exploration methodology. Section 4.4 explains the application and platform models used as inputs by our exploration tool. Section 4.5 explains the design exploration in detail. Section 4.6 presents the results. Section 4.7 gives an overview of the related work and section 4.8 presents the conclusions.

4.2 Coupling effects in mapping exploration

Previous section explained the complexity of developing efficient software for nomadic devices and their heterogeneous compute infrastructure. This complexity is due to the fact that when mapping software on hardware both memory and communication aspects need to be taken into account. Therefore we introduced the notions of horizontal and vertical coupling.

4.2.1 Horizontal coupling: Memory-aware task scheduling on processing elements

On platforms with multi-level memory hierarchies (e.g., L1 and L2 caches or scratchpads), the execution time of a computational task depends not only on the processor it is mapped onto and the corresponding power mode of the processor but also on the location of data in the memory hierarchy (e.g. in L1 or main memories). When mapping software tasks onto hardware in an energy efficient way, both the timing deadlines of the application as well as the resource constraints of the hardware have to be respected. If the mapping is split into different sub-problems and solved independently, it leads to sub-optimal solutions. To illustrate the problem, suppose that task mapping is done before memory exploration. This means that a decision has to be made what software task runs on what hardware processing element (spatial assignment). Since the location of data is not known yet but tasks need to meet their timing deadlines, the task-mapping exploration has to take one of two extreme assumptions:

1. All data read/writes are from/to the (slowest) main memory.
2. All input data is first brought into the (fastest) L1 memory before starting the computation and all output data is first produced in the L1 memory before being written off to the main memory.

The first assumption is too conservative and would unnecessarily force the tasks to run on faster and energy hungry processors or a higher power mode in order to guarantee meeting the deadlines, whereas the deadline could also be met if the data was put in the faster L1 memory while still running the task on the slower (energy efficient) processor or a lower power mode.

The second assumption is also undesirable because it creates unnecessary traffic due to copy operations between different memories. Even the data that is used only once is first copied to the L1 memory before being used. In order to efficiently utilize the small L1 memories, data objects that are not alive simultaneously may need to use the same memory space.

Both assumptions illustrate the fundamental problem that doing task allocation before data allocation leads to sub-optimal solutions. Whereas, the alternative solution of first exploring the mapping of data to memory before considering task mapping leads to similar problems.

4.2.2 Vertical coupling: Scratchpad aware dimensioning of buffer sizes

In [14], the authors show that scratchpad memories (SPM) consume on average 40% less energy and 46% less *area-time*[2] when compared to caches of the same capacity. Moreover, they provide much better timing predictability than caches. Therefore, SPMs are included in many embedded platforms such as ARM 10E, IBM Cell BE, GeForce GTX and Texas Instruments TMS370CX7X. Unlike caches that are managed by hardware and that select their contents on the principle of spatio-temporal locality, in SPM based systems the software is responsible for

[2]The product of chip area occupied by a memory and the memory access latency, used as a metric to evaluate memory designs

the allocation to scratchpads. Two important challenges for mapping dataflow programs on scratchpad based MPSoCs are *buffer dimensioning* and *scratchpad allocation*.

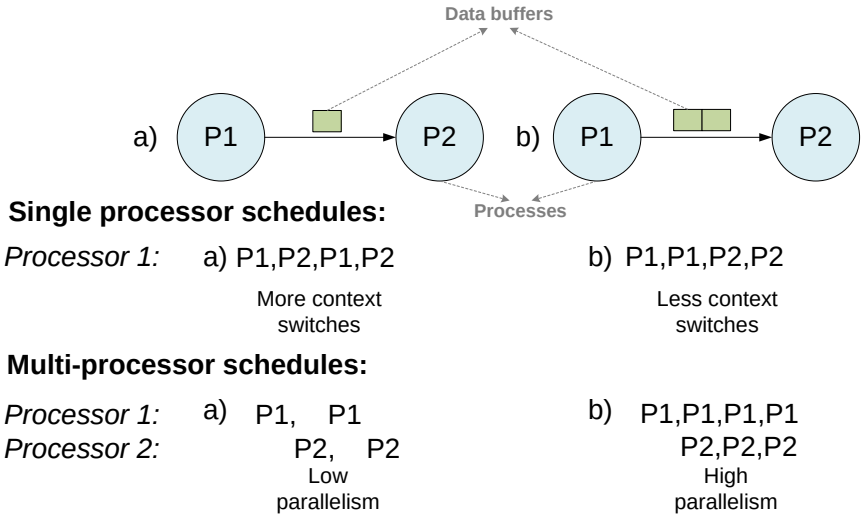


Figure 4.2: Buffer size trade-off with context switches and parallelism

Buffer dimensioning

Buffer dimensioning is deciding the sizes of the data buffers between computational processes over intervals of time. Figure 4.2 illustrates two ways in which buffer sizes directly influence performance of a dataflow program. P1 and P2 are processes of a dataflow program, (a) and (b) are two buffer dimensioning scenarios. In scenario (a) the buffer size between P1 and P2 is limited to 1 unit, in scenario (b) it is relaxed to two units.

- Consider the single processor schedules for these scenarios. Assume that the scheduler tries to minimize context switches while respecting buffer size restrictions. The number of context switches in scenario (a) are twice as compared to scenario (b) because the larger buffer allows the construction of schedules with lesser context switches.

- Consider the multi-processor schedules on a system with two processors. Assume that the scheduler tries to increase throughput by executing processes in parallel. In scenario (a), P1 and P2 can not execute in parallel due to lack of space in the data buffer. Whereas, in (b) they can execute in parallel, resulting in a higher throughput.

Larger buffer sizes provide the necessary decoupling required to reduce context switches and exploit parallelism. For a scratchpad based MPSoC, the execution time and energy consumption of a process[3] depends also on the type of memory its data buffers are mapped onto. Since large buffers may not fit into the fast but small scratchpads it is important to take these trade-offs into account.

Scratchpad allocation

Scratchpad allocation of a dataflow program is the selection of buffers mapped onto the scratchpad memory over different periods of time. *Data reuse* is the frequency with which data objects are reused. In a uni-processor system with only one execution path, allocating buffers with the highest data reuse to the scratchpad improves both the energy efficiency and the execution time of the application. However, this is not always true for multi-processor systems.

Consider the example in figure 4.3. The processes P2 and P3 are mapped onto different processors, creating two execution paths in the system. The longest execution path is marked red and the buffer with the highest data reuse is colored blue. Allocating the buffer with the maximum data reuse to the scratchpad has the maximum benefit for the energy efficiency, however, only allocating buffers in the longest execution path decreases the execution time of the application. Therefore in a multi-processor system it is sometimes necessary to allocate buffers in the longest execution path with lesser data reuse to the scratchpads in order to meet timing requirements.

[3] In this chapter we use the words process and actor for the nodes of a dataflow graph interchangeably

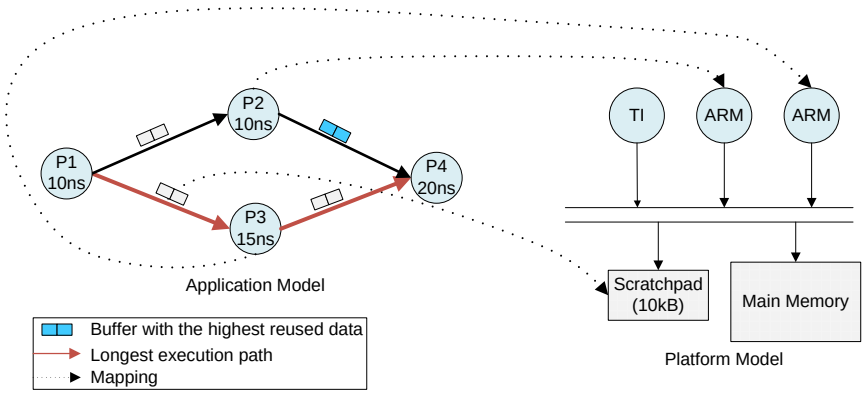


Figure 4.3: Scratchpad allocation for a multi-processor system

The two decisions (buffer dimensioning and scratchpad allocation) are interdependent. On the one hand, buffer sizes are required in order to make scratchpad allocation decisions, while on the other hand, the optimal size of a buffer depends on whether it is allocated in the main memory or the scratchpad. Furthermore, these decisions also depend on the allocation and scheduling of computation and communication.

4.2.3 Putting it all together

The examples illustrated in this section demonstrate fundamental interdependencies in different aspects of mapping software applications to MPSoC platforms. Figure 4.4 shows an abstract view of these interdependencies. We see that the optimal scratchpad allocations, buffer sizes, data transfers, processor allocations, context switches, parallelism and power modes; all depend on each other.

This illustrates the fundamental problem that the decisions of whether or not to copy data into local memories or to run tasks on different processors in order to save time and energy depend not only on each other but also on the availability of communication resources. To address this problem we propose a tightly coupled co-exploration technique that explores spatio-temporal schedules for computation, data storage

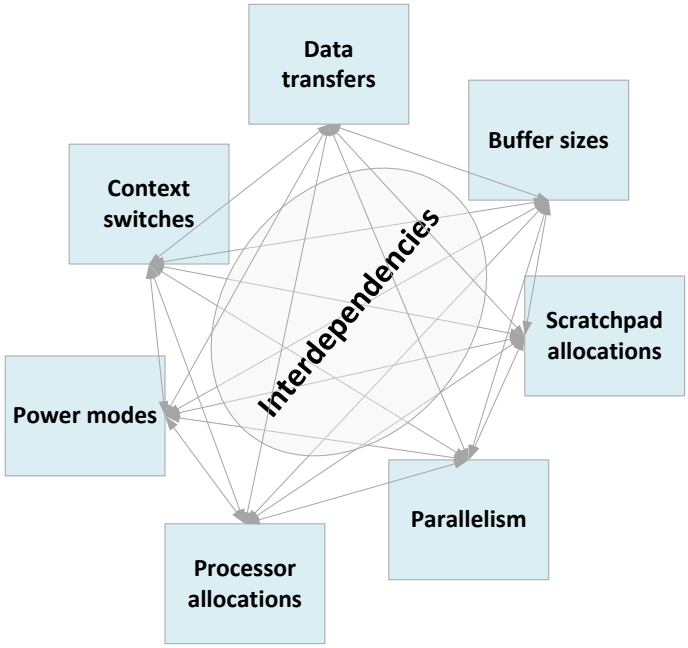


Figure 4.4: An abstract view of interdependencies between different aspects of mapping software applications to MPSoC platforms

and communication simultaneously considering the inter-dependencies between them, in order to find globally optimized solutions.

4.3 Overview of the mapping methodology

The previous section described inter-dependencies between different aspects of mapping applications to MPSoCs. This section gives an overview of the mapping methodology. It shows that developing software that needs to make such trade-offs between energy dissipation and throughput is very complex. To help a software engineer develop such software we introduce a mapping exploration methodology that finds Pareto-optimal mapping solutions that allows a runtime manager to make trade-offs between energy dissipation and throughput. Figure 4.5 illustrates the main steps in the methodology starting with the

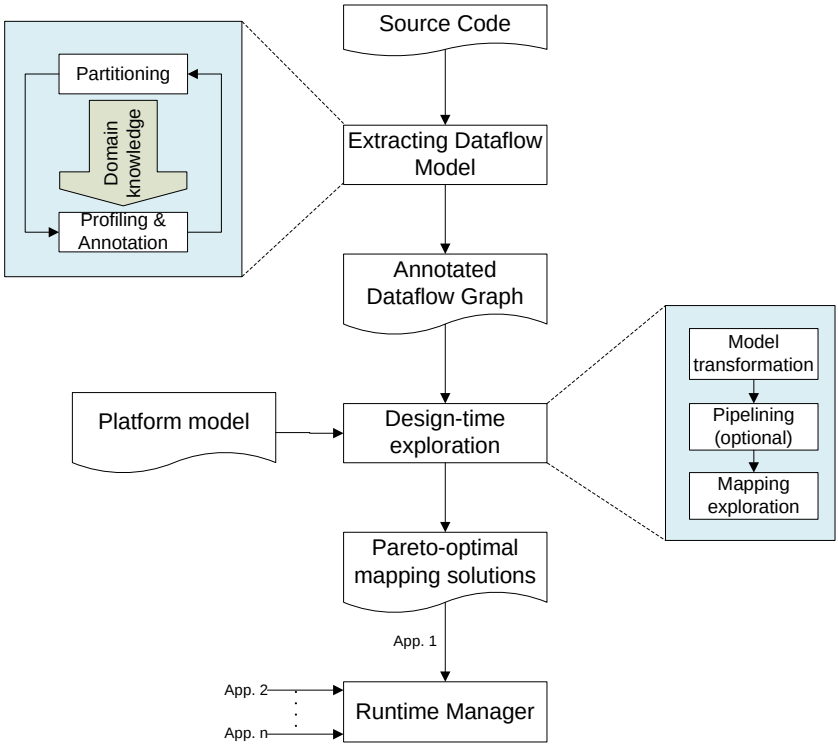


Figure 4.5: Overview illustrating different steps in the methodology.

application source code and ending with mapping solutions that are used by a runtime manager to schedule the applications.

The design-time mapping and scheduling exploration tool requires Annotated Synchronous Dataflow Graph (ASDFG). For new software this is not really a problem. Model Driven Design (MDD) tools such as Ptolemy [32] and UML MARTE [133] start by creating a dataflow graph of the application first and then use this model to generate source code. This dataflow graph can be annotated with profiling information to generate an ASDFG. However, this approach does not work for legacy applications. For legacy applications program analysis techniques need to be applied in order to extract a dataflow model. These program analysis techniques are either static or dynamic, each having their own

advantages and disadvantages.

- *Static program analysis* techniques for extracting dataflow graphs employ source code analysis tools such as, PN [132] and Pluto [19]. These tools are usually based on polyhedral models, that provide abstraction to reason about transformations on loop nests by modeling iterations of different statements and the dependencies between them in the form of polyhedrons. However, the downside of these approaches is that they do not support non-affine accesses or pointer arithmetic. Hence, major parts of the application often need to be re-written before the tools may be applied. Other tools that facilitate this re-writing effort have also been developed, e.g. CleanC [93] that point out the parts of the code that have to be changed, without doing the actual rewriting.
- *Dynamic program analysis* techniques for extracting dataflow graphs employ communication profiling tools such as, Pin-Comm [67]. These tools measure data communication between different parts of the application. An initial partitioning of the application is taken as a starting point for this technique. This initial partitioning is then iteratively refined through cycles of profiling and re-partitioning. The major drawback of this approach is the strong requirement of domain knowledge in order to guide the partitioning and re-partitioning.

Using the ASDFG the design time exploration tool explores the schedules of tasks over multiple processors, data buffers in different types of memories and data transfers over the communication bus. The next step is model transformation. In order to simplify the scheduling, we limit the concurrency of the ASDFG by unfolding it into a Directed Acyclic Task graph (DAG), as shown in figure 4.6. This DAG is then used for exploring the required schedules taking into account the horizontal and vertical dependencies.

The scheduling exploration is realized with IBM ILOG [78] where the mapping is modeled as a constraint based scheduling problem in Optimization Programming Language (OPL). The execution semantics of the DAG, along with the platform resources (processing, memory) are

modeled as constraints. The solver is then asked to find a minimum energy schedule for a given deadline. A set of schedules that give a trade-off between energy and execution time is found by repeating the procedure for the different deadlines. The run-time manager [143] uses these sets of Pareto-optimal schedules for scheduling the application at run-time. The run-time manager takes the states of all the applications deployed on the platform into account while selecting a particular configuration.

4.4 Platform and Application Models

This section describes the models for the application and the platform that are used as an input for the design-time exploration tool.

4.4.1 Platform Model

Most of the information about the platform required is extracted through detailed profiling of the application on the platform. A platform is described as a tuple *(Processors, Memories)*, where *Processors* and *Memories* represent the sets of processors and memories in the platform. A processor is defined as a tuple *(ProcessorID, CtxSwthTime, CtxSwthEnergy)*, where *ProcessorID* is a unique number for every processor, *CtxSwthTime* and *CtxSwthEnergy* are the time and energy consumed during a context switch. A context switch includes the loading and the initialization of the actor code before it can start processing data. A memory is defined as a tuple *(MemID, Size)*, where *MemID* is a unique identifier for every memory and *Size* is the size of the memory. The energy values and execution time of the context switch include those spent in the memories and interconnect along with those of the processor.

4.4.2 Annotated Synchronous Dataflow Graph

In the rest of this chapter we use Annotated Synchronous Dataflow Graph (ASDFG) as an abstraction for the application. An Annotated Synchronous Dataflow Graph (ASDFG) is a Synchronous Dataflow Graph

annotated with profiling information. The ASDFG is described as a tuple (A, C), where A is the set of *Actors* and C is the set of *Channels*.

Actors

An Actor is assumed to be a deterministic piece of code. In each execution instance it consumes a fixed amount of data from each of its input channels and produces a fixed amount of data on its outputs. The execution time and energy consumption of the actor depends only on the type of processor it is executed on and the memories where its input/output is mapped onto. In case when these properties also depend on the input data, worst case assumptions are taken. The actors themselves are stateless. Any required state is explicitly represented as a self loop channel. Actors are described as a tuple *(ActorID, Ports, Modes)*, where *ActorID* is a unique number for every actor. *Ports* is the set of all input and output ports of the actor. *Modes* is the set of all possible execution configurations for the actor, i.e. one configuration for each possible combination of processors and memories.

Modes

A mode is a configuration in which an actor can execute. Consider the application and the platform shown in figure 4.3; the actor $P2$ can be executed on either of the two processors. It has two ports each of which can be mapped to read/write the data either from/to the main memory or the scratchpad. Therefore, the actor $P2$ has eight modes. A mode describes the execution configuration as a tuple *(ModeID, ProcessorID, PortMemIDs, ExecTime, Energy)*, where *ModeID* is a unique number for every mode. *ProcessorID* identifies the processor used in this configuration. *PortMemIDs* is the set of *MemID* that identifies the memories used for each port, i.e. one *MemID* for each port. *ExecTime* and *Energy* are the execution time and energy consumed if the actor is executed in the given mode. These energy and execution time values are obtained through profiling and include the time and energy spent in the processors, memories and the interconnect.

Channels

A channel is defined as *(ChID, Source, Sink, InRate, OutRate)*, where *ChID* is a unique identifier for each channel. *Source* and *Sinks* are *ActorID*s of the source and sink actors connected to the channel. *InRate* and *OutRate* are the amounts of data produced or consumed by the source and sink actors respectively, each time they execute.

4.4.3 Model transformation to a task graph

A Synchronous Dataflow Graph is an auto-concurrent model of computation where parallelism is implicit. In order to derive an optimal schedule on a multiprocessor, all possible combinations of the different instances of actors need to be considered which is often not possible. A common approach is to first construct a Directed Acyclic Graph (DAG) where parallelism is explicit [102]. Figure 4.6 illustrates the model transformation of a synthetic SDFG. In order to construct a DAG from an SDFG, a periodically admissible sequential schedule (PASS) needs to be computed first. To compute a PASS balance equations for each channel are formulated and an integral vector is found that solves the system of equations. The PASS is then unfolded by an 'unroll factor' and dependencies are added between all the different instances of the actors [72]. The 'unroll factor' depends on the target platform and can be specified by the developer.

A DAG is defined as (T, E) where, T is a set of *Tasks* and E is a set of *Edges*.

Tasks

Tasks are instances of SDFG actors. Therefore they have all of the same properties and modes as the SDFG actor they belong to. It should be noted that if the tasks belonging to the same actor are executed one after another on the same processor, there is no context switch because the same actor is fired multiple times. Tasks are defined as *(TaskID, ActorID, Ports, Modes)*.

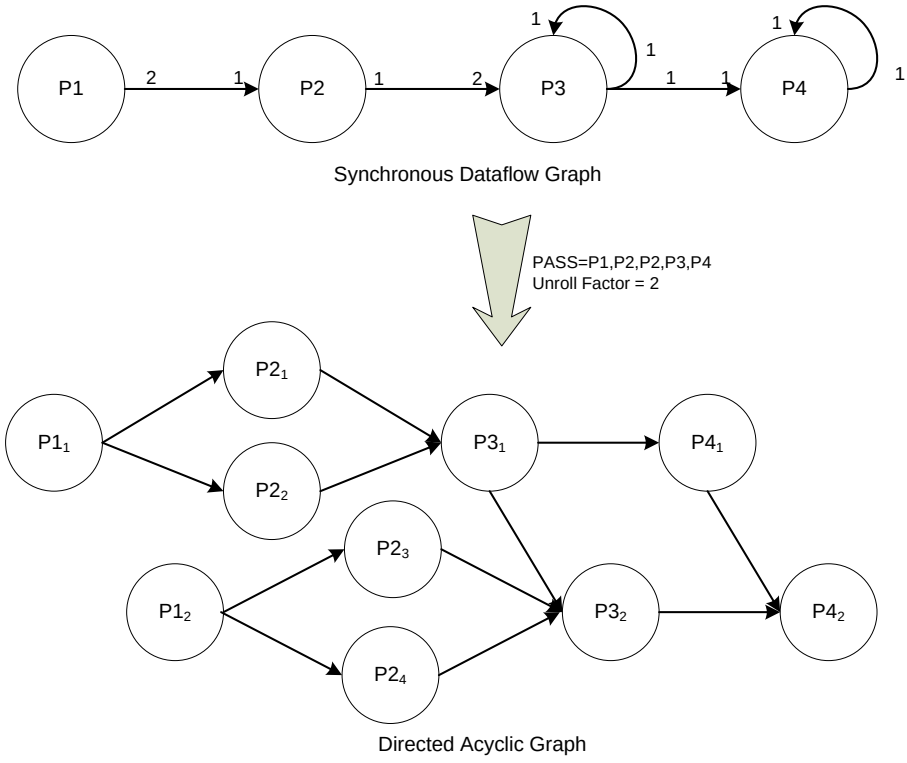


Figure 4.6: Model Transformation

Tasks of the same application can depend on each other. When executed they consume data from each incoming edge and produce data on the outgoing edges. The execution time and energy consumption of the task depends on three different types of parameters; (1) type of the processor it is executed on, (2) power mode of the processor, and (3) the memory mappings of the incoming and outgoing edges (e.g. L1 memory or main memory). These three parameters together form a *task-mapping*. The energy consumption, execution time along with the bus bandwidth required for each of the task-mappings are profiled and annotated to the tasks. Worst case estimates are taken in case of dynamism.

Degree of freedom in task-mappings: Assume that a task can be executed on P different types of processors, with M power modes each, the task has a total of E incoming and outgoing edges, each of which can be mapped onto L different types of memories. It then has $P \times M \times (L)^E$ possible task-mappings. Although the number of task-mappings grows exponentially with the number of edges connected, it usually does not cause a explosion of combinations, as the number of edges connected to a task is fairly small for many applications [127].

Edges

Edges are data transfers between the tasks. An edge is defined as *(EdgeID, Source, Sink, Size)*. It is possible that several edges in the DAG belong to the same channel in the SDFG. The *Size* of an edge is the greatest common divisor of *InRate* and *OutRate* of the SDFG channel it belongs to. The question of time dependent buffer sizes is now transformed into the question of the number of live edges in the DAG and their fixed sizes. Edges represent dataflow dependencies between tasks. They are annotated with the amount of data being transferred along the edge. Worst case estimates are again used in case of dynamism.

Degree of freedom in edge-mappings: An edge can be mapped to the memory and communication resources in five different ways:

1. *Local memory:* An edge can only be mapped onto the local memory of a processor if both the source and destination tasks of the edge are mapped onto the same processor. In this case data is produced and consumed in local memory; hence, no data transfers on the bus are required.

2. *Main memory:* If an edge is mapped onto the shared main memory, tasks will read/write data from/to the main memory as they execute. Therefore tasks require bus bandwidth to be allocated to them as they execute.

3. *Main-local:* The source task produces data in main memory, whereas the destination task consumes it from local memory.

Bandwidth allocation for two different types of data transfers on the bus are required for this mapping, (1) bandwidth allocation for the source task, and (2) bandwidth allocation for copying data from the shared main memory to local memory.

4. *Local-main:* The source task produces data in local memory, whereas the destination task consumes it from main memory. Bandwidth allocation for two different types of data transfers on the bus are required for this mapping, (1) bandwidth allocation for the destination task, and (2) bandwidth allocation for copying data from local memory to main memory.

5. *Local-main-local:* Both source and destination tasks produce and consume data in their local memories, however, they are mapped onto different processors. Therefore data needs to be copied from one local memory to main memory and then from main memory to the other local memory. Bandwidth is allocated for the two copy operations; however, no bandwidth is required for the edge during the execution of the tasks it connects.

4.5 Co-Exploration

In order to find a schedule for tasks on the processors and schedule bus bandwidth for these tasks (if necessary) at the same time schedule memory space and bus bandwidth for the edges, we formulate the problem as a constraint based scheduling problem and solve it with constraint solvers (e.g. IBM ILOG [78], GECODE [56]). Constraint solvers explores a search space for an optimal solution (i.e. solution with the minimum cost), such that all the constraints are satisfied.

In this section we describe the search space, constraints, minimization objective of the problem and the exploration procedure for the energy-speed trade-off. Figure 4.7 gives an overview of the co-exploration using a simple example with three tasks and two edges between them.

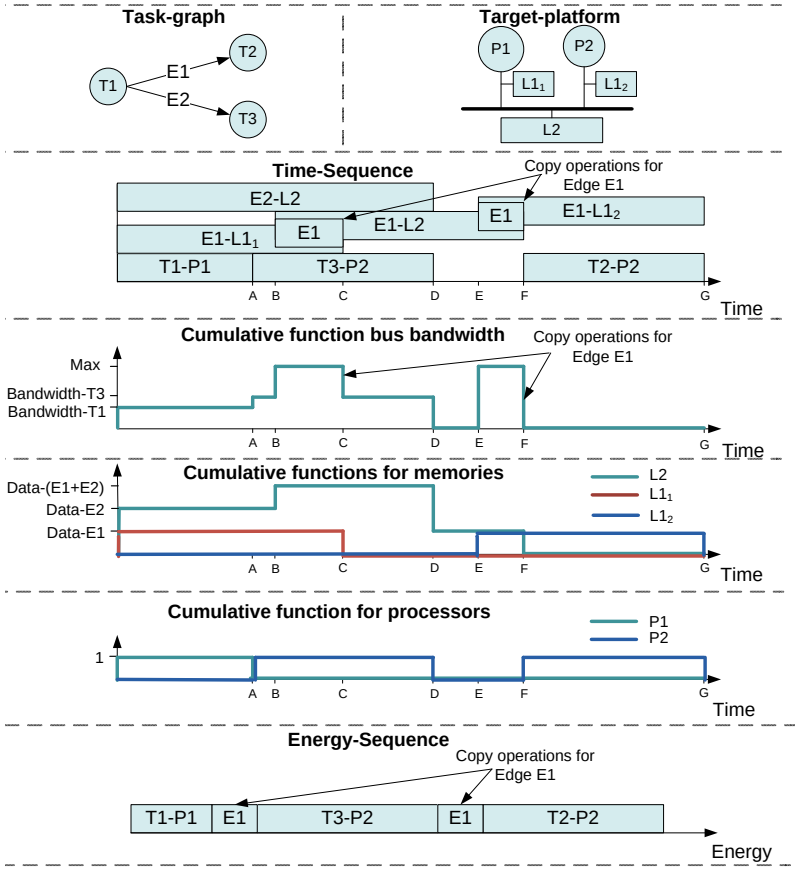


Figure 4.7: An example task-graph, a possible schedules for its resource usage and the corresponding cumulative memory usage on the target platform. Edge E1 is mapped as *local-main-local* and edge E2 as *main-memory*. The interval label T1-P1 indicated that the task T1 is executed on the processor P1.

4.5.1 Search Space

We have two types of activities in our problem: tasks and edges. These activities use five types of resources: processors, memory space, bus bandwidth, time and energy. The usage of these resources by the activities

form the schedules that define the solution.

The resources in our system are classified into two types; *Renewable* and *Nonrenewable*. Renewable resources are resources that can be returned to the system once a task is finished, such as processors and memories. The nonrenewable resources are permanently consumed, such as energy and time. Activities are modeled with the variables of type *Intervals*, whereas the usage of nonrenewable resources as *Sequences* of *Intervals* and renewable resources are represented as *Cumulative functions* of intervals over time.

Intervals

We use interval variables to model the consumption of non-renewable resources (energy and time) for a possible *mapping* of an activity. An Interval variable $\underline{a}$ has a start $s(\underline{a})$ and an end $e(\underline{a})$ when it is present; these variables can also be declared as 'optional' in which case they can also be absent $\perp$ i.e. they don't have a start or end. The domain of $\underline{a}$ is $dom(\underline{a})$:

$$dom(\underline{a}) = \{\perp\} \cup \{[s(\underline{a}), e(\underline{a}))|s(\underline{a}), e(\underline{a}) \in \mathbb{Z}, s(\underline{a}) \leq e(\underline{a})\}$$

The size of interval $\underline{a}$ is $IntervalSize(\underline{a})$:

$$IntervalSize(\underline{a}) = e(\underline{a}) - s(\underline{a})$$

.

Table 4.1 lists selected interval variables used in the model. Tasks are represented by two intervals, a time-interval and an energy-interval. The size of the *time-interval* equals the execution time of the task, including the time spent in memory accesses, for a particular possible mapping of the task. Similarly the size of the *energy-interval* represents the total energy consumed by the task. these are not optional intervals. Therefore each task must be allocated at least one time slot. The size of this interval is not fixed and depends on the selected mode. A *task mode* represents a particular mapping of a task i.e. the processor it is executed upon and memories it reads/writes its data. Every task mode has two interval variables associated with it, these are optional intervals and are absent if the mode is not selected.

Name	Optional	Description	Size
TaskModeTime[total no. of modes for all tasks]	✔	An array of interval variables with one interval for every mode of all tasks	Equals the profiled task execution time for the specific mode
TaskModeEnergy[total no. of modes for all tasks]	✔	An array of interval variables with one interval for every mode of all tasks	Equals the profiled energy dissipation for the specific mode
TaskTime[no. of tasks]	✘	An array of interval variables that represents the time for the execution of tasks	Equals the size of *TaskModeTime* of the selected mode
TaskEnergy[no. of tasks]	✘	An array of interval variables that represents the energy dissipation for tasks	Equals the size of *TaskModeEnergy* of the selected mode
EdgeTime[no. of edges]	✘	An array of interval variables that represents the overall span of the edge	From the start of the source task to the end of the destination task
EdgeMode[no. of edges][no. of modes]	✔	A two dimensional matrix of optional intervals. The interval *EdgeMode*[E,M] is present if the edge E is connected to the task of mode M and the mode M is selected	Equal the size of the *EdgeTime* interval if present, absent otherwise

Table 4.1: List of selected interval variables used in the model

The *EdgeTime* interval variable represents the overall life span of an edge. Depending how the edges are mapped (i.e. according to one of the five degrees of freedom we differentiated earlier) the span of an edge may be sub divided into different sub-activities. The two dimensional matrix

EdgeMode models the modes of the edges. Figure 4.7 shows the edge $E1$ mapped as a *local-main-local* edge and comprises of the following five sub-activities:

1. Initial storage in a $L1_1$
2. A copy operation to the $L2$
3. Intermediate storage in $L2$
4. A copy operation to the $L1_2$
5. Final storage in the $L1_2$

Other types of edge mapping only require a subset of these five sub-activities.

Sequences

A sequence is a total ordered set of interval variables. These interval variables can also have types assigned to them. An additional constraint of *noOverlap* on sequences can force all the intervals in a sequence to be non-overlapping. Optionally the intervals in a sequence can have *Types*. These types can be used to impose a minimum transition distance between the intervals of a sequence in the presence of a *noOverlap* constraint. This translates into

$$noOverlap(\pi, M) \Leftrightarrow \forall \underline{a}, \underline{b} \in \underline{A},$$

$$((\pi(\underline{b}) = \pi(\underline{a}) + 1) \Rightarrow (e(\underline{a}) + M(T(\pi, \underline{a}), T(\pi, \underline{b})) \leq s(\underline{b})))$$

where $\underline{A}$ is a set of intervals, π is a sequence in $\underline{A}$ with types T and M is a matrix with the minimum transition distances between the different types of intervals.

ProcessorTime is an array of non-overlapping sequences, one for every processor. The domain of these sequences is the array of time intervals of task modes *TaskModeTime*, for all modes that use the particular processor. In the sequence every *TaskModeTime* interval has a *Type*

that represents the SDFG actor to which the task belongs *ActorID*. A minimum transition distance that equals *CtxSwitchTime* of the processor is imposed whenever tasks belonging to different actors are executed consecutively on the same processor, as shown in figure 4.8.

ProcessorEnergy is an array of non-overlapping sequences, one for every processor. The domain of these sequences is the array of energy intervals of task modes *TaskModeEnergy*, for all modes that use the particular processor. In the sequence every *TaskModeTime* interval has a *Type* that represents the SDFG actor to which the task belongs *ActorID*. A minimum transition distance that equals *CtxSwitchEnergy* of the processor is imposed whenever tasks belonging to different actors are executed consecutively on the same processor, as shown in figure 4.8.

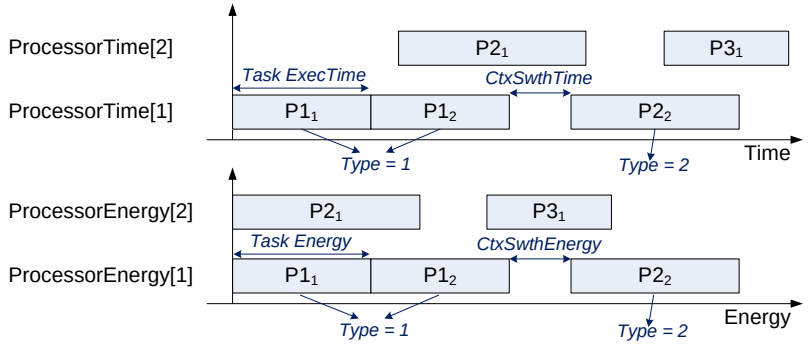


Figure 4.8: ProcessorTime & ProcessorEnergy sequences

Activities such as parallel processing of tasks onto different processors or interleaved copy operations between the local and main memories can happen simultaneously. Therefore, time-intervals of activities are allowed to overlap with each other as long as they respect the capacity constraints of the platform and execution semantics of the model of computation.

Cumulative functions

Cumulative functions are used to represent the usage of renewable resources, such as the usage of different memories. These functions can

be composed of elementary functions such as *Pulse(height,interval)* and *StepAtStart(height,interval)* shown in figure 4.9. A cumulative function

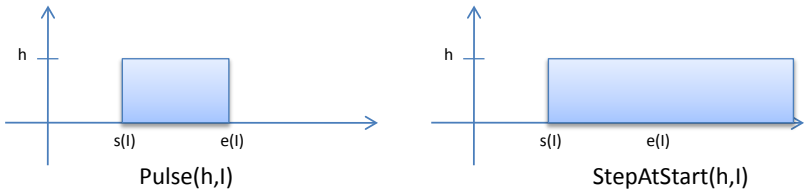


Figure 4.9: Elementary Functions

for a the usage of a renewable resource is the sum of pulse functions over time-intervals of all the activities using that resource i.e there is a separate cumulative function representing the usage of each processor, memory and interconnect. $f(R)$ is the cumulative function for the resource R.

$$f(R) = \sum_{\forall A | R_A \neq 0} R_A \times \Pi(I_A)$$

R_A is the resource usage of resource R by the activity A (the amount of data in the case of memories, the bandwidth for interconnect and a binary 0/1 for processors). $\{A|R_A \neq 0\}$ is the set of all activities or sub-activities that use resource R. $\Pi(I_A)$ is a pulse function on the time-interval of the (sub-)activity

MemUsage is a set cumulative functions that represents the memory usage of different memories over time. *MemUsage[Mem]* is a model of the usage of the memory *Mem*.

$$MemUsage[Mem] = \sum_{\{\forall E \in Edges | EdgeMode[E,Mem] \neq \perp\}} Pulse(E.Size, EdgeTime(E))$$

4.5.2 Constraints

A valid mapping solution needs to satisfy several constraints. These constraints impose boundaries within the search space, thus limiting

the search space. Figure 4.10 gives a general overview of the constraints and costs relationships between the different activities and resources. Activities and resources are represented as ovals and rectangles respectively and the cost or constraint relationships between them are represented as arrows. In the remainder of this section we will explain these cost and constraint relationships.

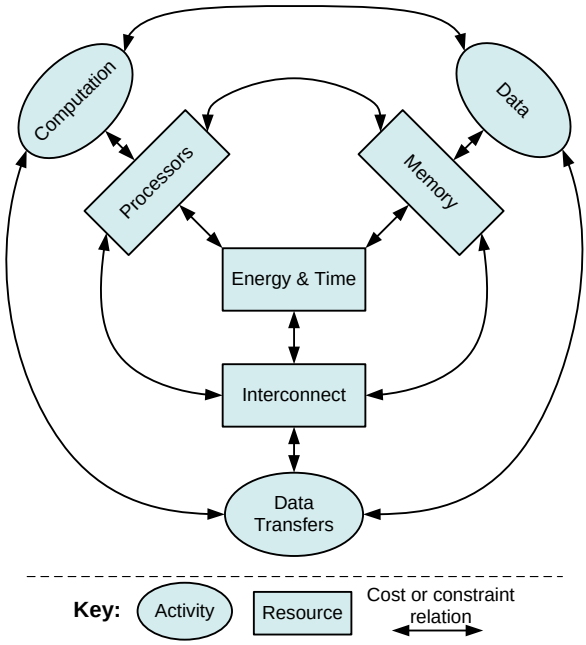


Figure 4.10: Relations between activities and resources

Constraints between activities and resources

- The cumulative functions for all the memories are always less than, or equal to, the size of the memories.

$$\forall_{Mem \in Memories},$$

$$MemUsage[Mem] \leq Mem.Size$$

Similarly, the cumulative function for the interconnect is always less than, or equal to, the arbitrated net bandwidth[4] and the cumulative functions for all the processors are always less than, or equal to, one.

- One and only one *TaskModeTime* interval is present for every task in a valid schedule, and the *TaskTime* interval starts and ends with the selected *TaskModeTime* interval. This behavior is captured by the *alternative* constraint in OPL. An $alternative(\underline{a}, \{b1, .., bn\})$ constraint implies that, if the interval $\underline{a}$ is present then exactly one of $\{b1, .., bn\}$ is present. And that, $\underline{a}$ starts and ends with the chosen interval.

- If a task mode M is chosen the interval $TaskModeTime[M]$ is present, the corresponding interval $TaskModeEnergy[M]$ also has to be present and the sequences *ProcessorTime* and *ProcessorEnergy* should have the same order:

$$\forall_{M \in Modes},$$
$$presenceOf(TaskModeTime[M]) \Rightarrow$$
$$presenceOf(TaskModeEnergy[M]).$$
$$\forall_{M \in Modes, K \in Modes | M.ProcessorID = K.ProcessorID},$$
$$s(TaskModeTime[M]) \leq s(TaskModeTime[K]) \Rightarrow$$
$$s(TaskModeEnergy[M]) \leq s(TaskModeEnergy[K]).$$

Constraints between activities and activities

The execution semantics of the model of computation form the constraints between the different activities:

- Each task executes once and only once.

[4]The arbitrated net bandwidth is usually the physical bandwidth, depending on the arbitration policy and its latency and fairness guarantees.

- The edges enforce precedence constraints between the tasks and the life time of an edge starts at the end of its source task and ends at the end of its sink task:

$$\forall_{E \in Edges},$$
$$s(TaskTime[E.Sink]) \geq e(TaskTime[E.Source]),$$
$$s(EdgeTime[E]) = e(TaskTime[E.Source]),$$
$$e(EdgeTime[E]) = e(TaskTime[E.Sink]).$$

  Similarly, the time intervals of memory storage sub-activities start with the starts of its source task or copy operation and ends with its destination task or copy operation, as shown in figure 4.7.

- The interval $EdgeMode[E, M]$ is present if the edge E is connected to the task of mode M and the mode M is selected:

$$\forall_{E \in Edges, M \in Modes},$$
$$presenceOf(TaskModeTime[M]) \Rightarrow$$
$$s(EdgeMode[E, M]) = s(EdgeTime[E]),$$
$$e(EdgeMode[E, M]) = e(EdgeTime[E]),$$
$$\neg presenceOf(TaskModeTime[M]) \Rightarrow$$
$$\neg presenceOf(EdgeMode[E, M]).$$

Constraints between resources and resources

The platform model constitutes the constraints between resources. These constrains express the architecture of the platform; i.e which processors can use which memories and which communication requires the shared bus, as shown in figure 4.1.

4.5.3 Optimization objective and exploration

All tasks need to finish before the deadline; therefore a global constraint on the end times of all tasks is imposed:

$$\forall_{T \in Tasks} e(TaskTime[T]) \leq Deadline$$

The objective of the optimization is to minimize energy consumption while meeting the timing requirements and resource constraints. For our model we assume all energy spent in the processors and memories is used by tasks, in the form of *TaskModeEnergy* intervals. These intervals are aligned into *ProcessorEnergy* sequences, minimizing the total sum of the ends of these energy sequences will minimize the total energy consumption.

$$\textbf{minimize} \sum_{P \in Processors} ProcessorEnergy[P]$$

The deadline is iteratively increased giving a minimum energy schedule each time and the solutions that provide a trade-off on either the energy or time axis form the set of Pareto-optimal solutions. More sophisticated multi-objective exploration techniques [50, 51] that may converge faster, but the main focus of this thesis is on the quality of the solutions.

4.6 Evaluation

Two use cases are used for the evaluation of this approach. Vertical coupling in memory management is elaborated for an H.264 decoder use case and horizontal coupling results are presented for an image processing application, the cavity detector [26].

4.6.1 H.264 Decoder use-case

In this section we present the results of our technique for mapping an implementation of the H.264 decoder (figure 4.11). The H.264 decoder implementation takes a stream of H.264 and processes it frame by frame. Our TI OMAP 35xx like multiprocessor platform consists of

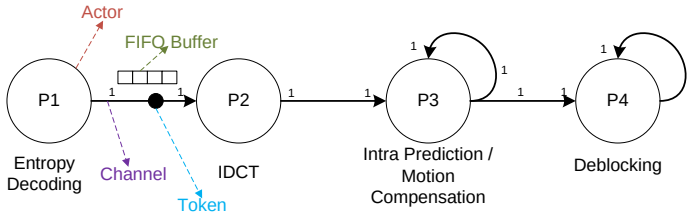


Figure 4.11: Synchronous Dataflow Graph for H.264 decoder

two RISC processors (Strong ARM 1100x) operating at 200MHz, one VLIW processor (TI-C64X+) operating at 500MHz a scratchpad memory (SRAM) of 64kB and a main memory (SDRAM) of 128MB. We profile the application for execution times using SimItARM and TI-CCStudio simulators for the StrongARM and TI-C64X+ processors respectively. The dataflow between the actors was measured using PinComm [67] and the memory accesses by Cachegrind. For the energy consumption, we use JouleTrack [114] and a functional level power analysis model of TI-C6X [81] for the StrongARM and TIC64X processors respectively, we choose these profilers because they have a good support for the processors used. These tools provide decent estimates with an error margin of less than 5%, this is sufficient for the study because both explorations (the baseline and our co-exploration) are based on the same profiling data. Some modern SoCs are equipped with built-in power metering sensors that provide highly accurate system-level energy measurements; we recommend using them if available instead of simulated energy values. For this experiment we profiled 15 different sample videos of resolution 352x480 for every frame, and took the worst case values for each actor, higher resolutions make the application even more memory constrained. An unroll factor of two was used for the ASDFG. A higher unroll factor might improve the quality of our results, but the amount of time required for the exploration would significantly increase.

Effects of varying scratchpad sizes

Figure 4.12 shows that varying the amount of scratchpad memory for the buffers significantly effects the energy-execution time trade-off. We

can observe that in order to meet the same timing deadline with a smaller scratchpad memory available, a higher energy schedule is often required. The scratchpad sizes affect several aspects of scheduling. Smaller scratchpads can cause the buffer sizes to shrink resulting in more context switches and less parallelism. Also, they can cause more DAG edges to be mapped onto the main memory thus increasing the execution time and energy consumption of the tasks connected to those edges. Furthermore, in order to meet the deadlines some tasks may have to be moved onto a faster but more energy hungry processor (in this case the TI-C64X+) thus causing higher energy schedules for the same timing performance.

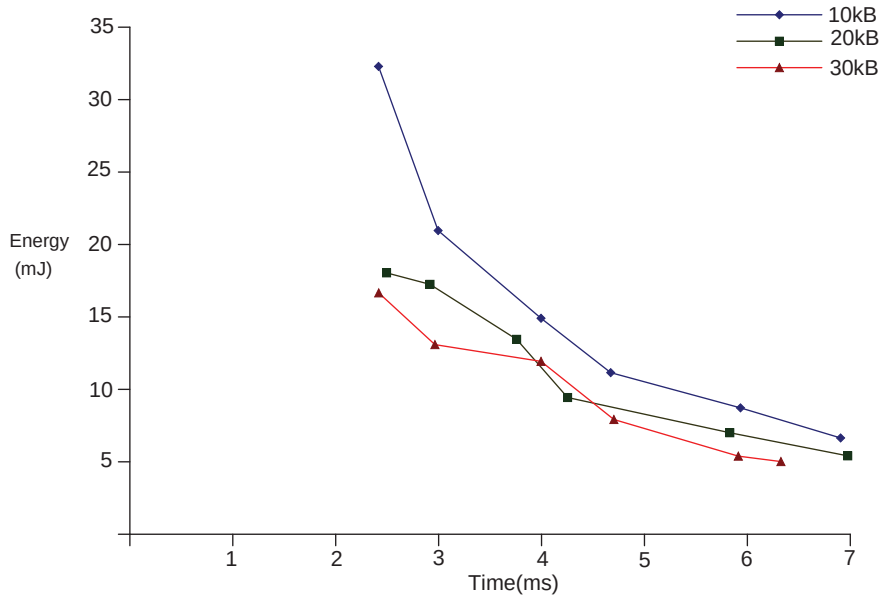


Figure 4.12: Energy throughput trade-off under varying scratchpad sizes

Memory sharing buffers versus baseline allocation

Efficient utilization of memory space is very important for a scratchpad based system. We study the effects of *memory sharing* optimization [52, 97] on the trade-off between throughput and execution time. In the memory sharing approach buffer sizes are allowed to vary over time. Therefore, the same memory space can be used by different buffers. In the baseline allocation approach buffer sizes remain static. We simulated the baseline approach by using a *StepAtStart* function instead of a *Pulse* function in the *MemUsage* function.

Figure 4.13 plots the time deadlines versus the energy consumption for the mapping solutions found by both, the baseline and our memory sharing approach. It can be seen that for every given deadline our approach finds mapping solutions that have significantly less energy consumption when compared to the baseline approach. It is important to note that a memory sharing implementation is expected to require more runtime de-fragmentation overhead compared to a baseline allocation. However, de-fragmentation effects are not modeled in this simulation.

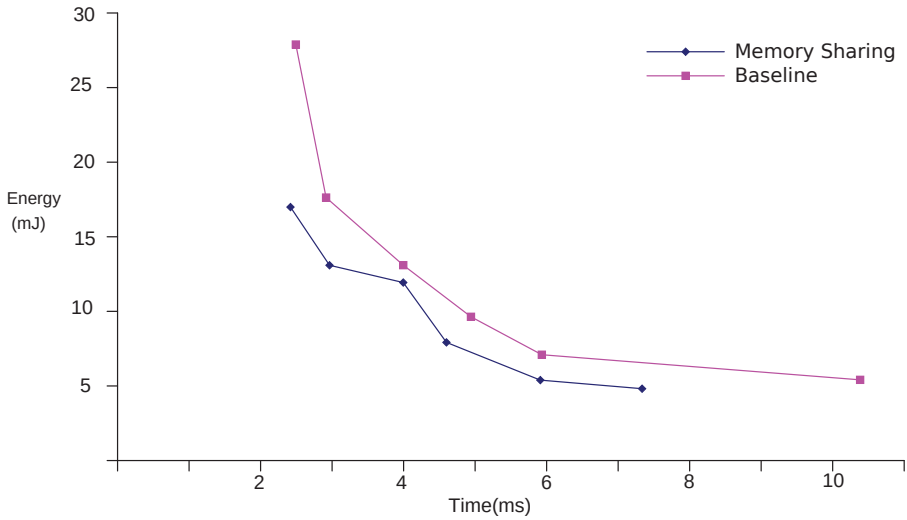


Figure 4.13: Memory sharing vs baseline allocation

4.6.2 Cavity detector use-case

To evaluate our co-exploration technique we study two aspects: the quality of the mapping solution and the scalability of the approach. To evaluate the quality of the mapping solution we mapped the pipelined version of the cavity detector application [26] on a heterogeneous MPSoC (see figure 4.14). $T1_1$ and $T1_2$ in the task graph shown in figure 4.14 are instances of the same function Horizontal Blur. The platform consists of four StrongARM 1100 and two TI-C64X+ processors (as shown in figure 4.14), each with a local L1 memory of 4kB. The platform also features a shared L2 memory of 128MB. A shared bus connects all memories and processors.

To profile the execution times of application tasks, SimItARM and TI-CCStudio simulators were used for the StrongARM and TI-C64X+ processors respectively. For the energy consumption, the energy models JouleTrack [114] and functional level power analysis model of TI-C64X+ [82] were used for the StrongARM and TI-C64X+ processors. The amount of communication between different tasks was measured with the architecture independent communication profiler PinComm [68].

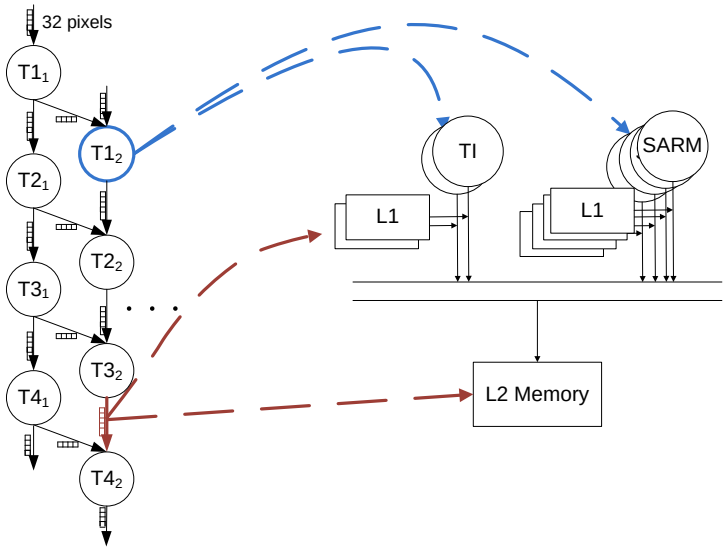


Figure 4.14: Mapping pipelined cavity detector on MPSoC

For evaluating the quality of the mapping solutions, the execution time and energy consumption of the schedules found by the co-exploration technique are compared with its decoupled counterpart with worst case assumptions. Figure 4.15 shows the results of the experiments. Each point in the trade-off space of figure 4.15, represents a unique mapping solution. The energy consumption and execution time are synthesized using the three schedules that constitute the mapping solution (computation, memory and communication). From figure 4.15 we observe that co-exploration can find significantly better mapping solutions (i.e. lower energy consumption for the same timing deadline) than the decoupled approach.

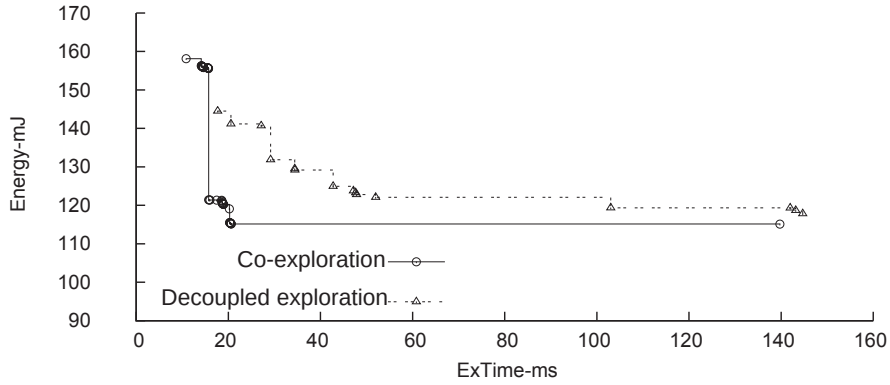


Figure 4.15: Exploration results for the different techniques

4.6.3 Scalability

In order to be applied to complex real life applications, the design-time co-exploration should be reasonably scalable. Two aspects of scalability of the co-exploration approach are studied:

1. Scalability with respect to an increasing number of processors on the platform.

2. Scalability with respect to an increasing number of tasks in the application.

Three different types of applications are used for this experiment: (1) synthetic application task graphs (generated by the TGFF [46]), (2) two different implementations of the cavity detector application [26] with different pipeline widths (CD-BF4 with a pipeline width of four, and CD-BF9 with a pipeline width of nine), and (3) an MP3 decoder implementation [79]. All these experiments were run on an Intel Core2Duo 2GHz machine. Table 4.2 show the results of the scalability experiment,

Application	**No.of Tasks**	**SARM**	**TI**	**Time (h:m)**	**Memory** (kB)
TGFF-1	22	4	2	1:32	25
TGFF-2	17	4	2	0:59	38
TGFF-3	13	4	2	0:43	70
MP3 decoder	25	4	2	1:29	25
CD-BF9	16	4	2	0:54	16
CD-BF9	36	4	2	1:49	100
CD-BF9	36	8	3	1:54	45
CD-BF9	36	12	4	1:50	41
CD-BF9	36	16	3	2:04	48
CD-BF9	36	20	4	1:58	48

Table 4.2: Co-Exploration Scalability

it is observed that the time taken for the solver does not increase exponentially when increasing the number of tasks or processors.

4.7 Related Work

A lot of work has been done on scheduling and mapping applications to MPSoC. For real-time systems, there are several static scheduling algorithms which can be used for the purpose of design-time exploration.

A good overview of classical scheduling algorithms can be found in [107], however, most of this work is limited to either uni-processor or homogeneous multiprocessors. The majority of the work only optimizes the performance of the application [11, 18], but the energy consumption is not addressed. We focus on heterogeneous MPSoCs and optimize energy consumption along with performance.

The problem of mapping applications onto heterogeneous platforms is addressed by [36, 104], but these solution are limited to mapping tasks on processors. Most of the literature considers only a subset of the three domains, either processing and communication exploration [49, 90], processing and memory exploration [20, 122], only communication exploration [12, 88], only memory exploration. In [28] constraint programming based approach is used, but unlike our approach they do the processing and memory exploration in a decoupled manner and do not consider multiple power modes or communication aspects.

Bio-inspired [50, 51] techniques have also been proposed for solving the mapping problem. An ant colony based multi-stage mapping technique is presented in [51], however, it only optimizes the execution time of the application ignoring the energy aspect. A multi-objective evolutionary technique considering both timing and energy aspects is presented in [50], however, mapping is defined only as an assignment problem and not as a scheduling problem. Our co-exploration technique provides allocation and scheduling of computation, communication and memory, taking into account the interdependancies between them.

The vertical coupling problem addressed in this chapter crosscuts two concerns in the domain of memory management for embedded systems, buffer dimensioning and Scratchpad allocation.

Buffer dimensioning

Several techniques are available for calculating the minimum buffer requirements such that there are no deadlocks [2, 59], while other calculate buffer requirements for *rate-optimal* schedules [65, 96]. Between these two extremes, other techniques explore trading of throughput and buffer size [119, 144, 147]. However, none of these techniques takes platforms with complex memory hierarchies into account, such as today's embedded devices. For such platforms the actual execution times of actors might depend on whether data is mapped onto a fast scratchpad memory or a slower flash memory. Therefore, the buffer size throughput trade-off also depends on the sizes of scratchpads, our co-exploration technique takes this interdependence into account.

The concept of letting buffers share the same memory space has been studied in [97] on a coarse grain level, where buffers with non-overlaying lifetimes can share the same memory space. In [52] on a fine grain level where even buffers with overlapping lifetimes are allowed to share the same memory space. These techniques have been shown to significantly reduce the overall memory requirements of streaming applications. However, our goal is to provide users with better energy efficiency and quality of service. Therefore we study the impact of such an optimization on the trade-off between execution time and energy.

Scratchpad allocation techniques

The problem of content selection for scratchpads has been extensively studied for C-like application models. [6] presents a survey of scratchpad allocation techniques. Most of these techniques suffer from the lack of information about the program structure [16] and are sometimes not applicable to applications with non-affine accesses, pointers, passing by reference, dynamic assignments etc. Therefore the source code usually needs to be 'cleaned' before these techniques could be applied [93]. In contrast to all these approaches we propose a scratchpad allocation technique at the level of dataflow graphs.

The literature on scratchpad allocation for dataflow graphs is quite limited [16, 17, 35]. In [35] a scratchpad aware scheduling technique is presented that maps both code and data segments of an application described as a dataflow diagram. The trade-off between context switches and buffer sizes is modeled but unlike our approach parallelism aspects and energy costs are not considered. The methodologies presented in [17] and [16], dynamically allocate code and data of a Hetrochronous Dataflow Graphs (HGFGs) to scratchpads in a Harvard architecture, however, in contrast to our approach they do not explore the buffer sizes.

4.8 Conclusion

This chapter presents a design-time co-exploration technique that schedules tasks on processors, data objects on the memories and data

transfers on the interconnect. The inter-dependencies between the three different types of schedules are modeled and accounted for, buffer dimensioning is scratchpad-aware and trade-offs with context switches and parallelism are explored. Vertical and horizontal decoupling are studied with the help of two use cases.

Validation on an image processing application (cavity detector) shows that the tightly coupled co-exploration technique that takes all the interdependancies into account is able to find mapping solutions that are significantly more energy efficient and/or faster when compared to decoupled exploration techniques. Whereas, our results for the H.264 decoder show that awareness of scratchpad sizes significantly affect the trade-off between energy and execution time and that letting buffers share the same memory space allows significantly better mappings on the energy-time trade-off for a scratchpad based system.

The next chapter applies dataflow based modeling and optimization techniques for the deployment and scheduling of applications in ubiquitous environments.

Chapter 5

Deployment and configuration of ubiquitous computing applications

The previous chapter discussed techniques for mapping real-time applications on parallel multi-core platforms. In this chapter we tailor the methodology for the domain of ubiquitous computing and use it for the deployment and configuration of component based applications on distributed systems[1]. Two use cases are presented in this chapter. The first use case explores the trade-offs for energy efficient deployment of artifacts in a wireless sensor network based (WSN) on the *Component and Policy Infrastructure* (CaPI) [91]. The second use case explores the deployment and configuration tradeoffs for a component based application: *fitness monitoring and fall detection.*

5.1 Introduction

Ubiquitous computing is a concept, where the presence of computing is everywhere and anywhere. From wearable mobile objects such as

[1]Parallel systems have a logically shared memory; whereas, distributed systems have logically (and often physically) distributed memories.

smart watches and smart shoes to fitted installations such as smart windows and smart doors, computing is *ubiquitous* and deeply embedded. Systems that run these ubiquitous computing applications are often highly distributed and very heterogeneous. Computing devices in such systems, vary from minute energy harvesting sensors and actuators with limited computation and communication capabilities over nomadic devises with moderate computation and communication abilities yet limited energy to stationary servers and infrastructures with adequate resources. Moreover, ubiquitous computing applications are often open-ended and dynamically interact with the environment or user. Furthermore, there are multiple often-conflicting optimization objectives that make deployment and configuration of these applications a challenging task, e.g., maximizing quality of service and battery life.

Sawyer et al. [109] present a constraint programming based approach for configuring wireless sensor networks (WSNs). Functional and quality-of-service requirements are modeled as a set of goals and the dependencies between them. Different configurations (i.e. hardware and software) may operationalize the same goals in different ways and a constraint solver is used to find the optimal configuration. In contrast, we use a dataflow-centric approach. By extracting a dataflow model of the application and mapping it onto an abstract model of the physical system we find the optimal[2] solutions for the deployment of the application and configuration of the hardware.

In order to evaluate our approach two use cases are used. The first use case explores the energy efficient deployment of software artifacts in a wireless sensor network, for four different representative scenarios. The second use case explores deployment and configuration trade-offs between the quality of service (error rate) and the CPU load for a fitness monitoring and fall detection application.

The remainder of the chapter is structured as follows. Section 5.2 presents two application use cases used for evaluating our approach. Section 5.3 gives an overview of the methodology. Section 5.4 discusses the results. Section 5.5 presents the conclusions.

[2]For a certain level of abstraction.

5.2 Use cases

The first use case explores the tradeoffs for energy efficient deployment of artifacts in a wireless sensor network, based on the *Component and Policy Infrastructure* (CaPI) [91]. The second use case deals with monitoring of user's general well-being and detection of alarming situations. The first use case demonstrates the applicability of our approach in distributed and heterogeneous systems. The second use case applies the methodology for optimizing a highly dynamic soft real-time application.

5.2.1 Energy-aware application deployment for a CaPI based WSNs

CaPI [91] is a *Component* and *Policy* based Infrastructure for Wireless Sensor Networks (WSNs). A WSN is a network of autonomous distributed sensors called "nodes" that monitor physical or environmental conditions such as temperature, pressure, light etc. The nodes are typically equipped with one or more sensors, a micro-controller and a radio frequency transceiver, and are usually powered by a small battery. Nodes usually run distributed software applications, communicating and collaborating with each other. CaPI provides two contemporary approaches for implementing software functionality on WSNs, *components* and *policies*. Components are software binaries with standardized interfaces that execute directly on the processor or a virtual machine. Policies are software scripts executed by a policy engine. The deployment and reconfiguration costs for components are higher than policies; however, they are generally more energy efficient.

Motes are nodes that gather data from fellow nodes in their vicinity and send it to a base station for further processing or storage. Motes are often better provisioned in terms of energy budget, computational power and/or transmission range. For a large wireless sensor network that consists of different types of sensors acting as motes or nodes, where software artifacts may be deployed as components or policies, the decision of which artifact to deploy and where can be complex. We use a dataflow inspired modeling and exploration techniques for finding the optimal deployment configuration.

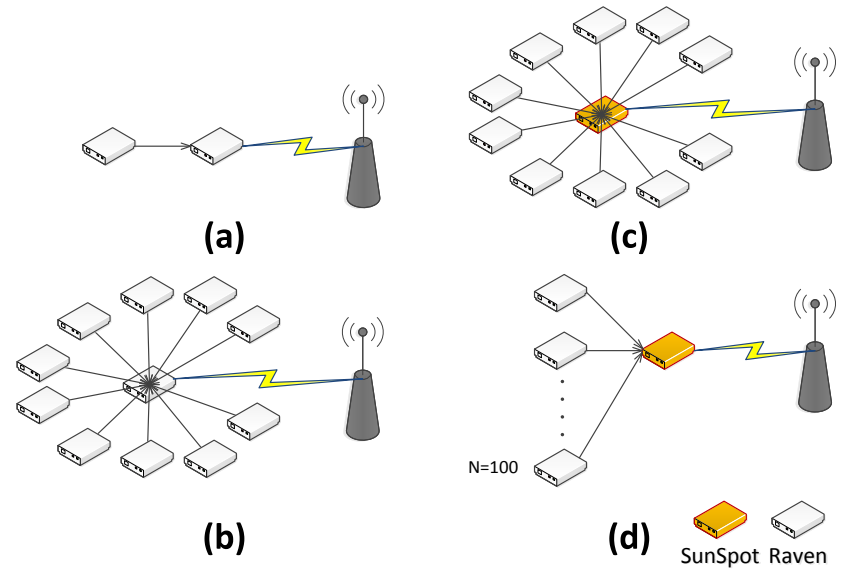


Figure 5.1: Illustrates four representative scenarios for the WSN use case: (a) One Raven node forwards its data to another Raven node, which then forwards it to a base station. (b) Ten Raven nodes are connected to one Raven node in the form of a star topology. The central node is acting as a mote; it collects data from the surrounding nodes and forwards it to the base station. (c) The central mote is a SunSpot, it collects data from ten Raven nodes and forwards it to the base station. (d) A SunSpot mote is collecting data from hundred Raven nodes and sends it to the base station

In this use case, the wireless sensor network consists of two types of nodes:

- **SunSPOTs**[3] are based on a 180 MHz AT91RM9200 processor with 512kB RAM and 4MB flash memories. The processor runs applications on top of a Java "Squawk" virtual machine. For communication, an IEEE 802.15.4 compliant radio transceiver,

[3]http://www.sunspotworld.com/docs/Yellow/SunSPOT-TheoryOfOperation.pdf

CC2420 from Texas Instruments is integrated. The radio consumes 61.2 mW (approx.) of power when switched on.

- **Ravens** are based on an Atmel 8-bit AVR RISC microcontroller running at 16MHz, with 128kB ISP flash memory and 16kB SRAM. An IEEE 802.15.4 compliant radio transceiver (AT86RF230) is embedded within the node. The processor requires 24 mW of power at 16 MHz and the radio again consumes 61.2 mW (approx) of power when switched on. Raven nodes are smaller, cheaper and require less power; whereas, the SunSPOTs have more computational power and memory.

In order to validate the applicability of our approach, we selected four representative scenarios, shown in figure 5.1. In the first scenario (a), one Raven node forwards its data to another Raven node, which then forwards it to a base station. In the second scenario (b), ten Raven nodes are connected to one Raven node in the form of a star topology. The central node is acting as a mote; it collects data from the surrounding nodes and forwards it to the base station. In the third scenario (c), the central mote is a SunSpot, it collects data from ten Raven nodes and forwards it to the base station. In the fourth scenario (d), a SunSpot mote collects data from hundred Raven nodes and forwards it to the base station. We explore the possibilities of deploying software artifacts (components or policies) for processing sensor data in the WSN. The artifact can be deployed either at a source node or at an intermediate node that forwards the data to the base station. The results of this exploration are provided in section 5.4.1.

5.2.2 Fitness monitoring and fall detection

Physical wellness and health are highly inter-linked. Mobility is seen as an essential decisive factor to maintain an altogether independent living. A detailed account of assisted living technologies and functions have been outlined by Sun et al. [123]. Ensuring the safety and security of the user with the help of alarms, monitoring the health and well-being of the user, and the use of interactive and virtual services to help support the user are just few of them. Hence, in this use case, the mobility of the user is

being monitored by inferring the physical activity of the user (standing, walking, number of steps taken, etc.). The system detects a fall in a smart way by not only relying on data provided by accelerometers, but also by incorporating knowledge about the location of the fall to infer the likelihood of the fall (for example, to reduce false positives due to dropping yourself in a chair) and to notify the caregiver in case of an emergency.

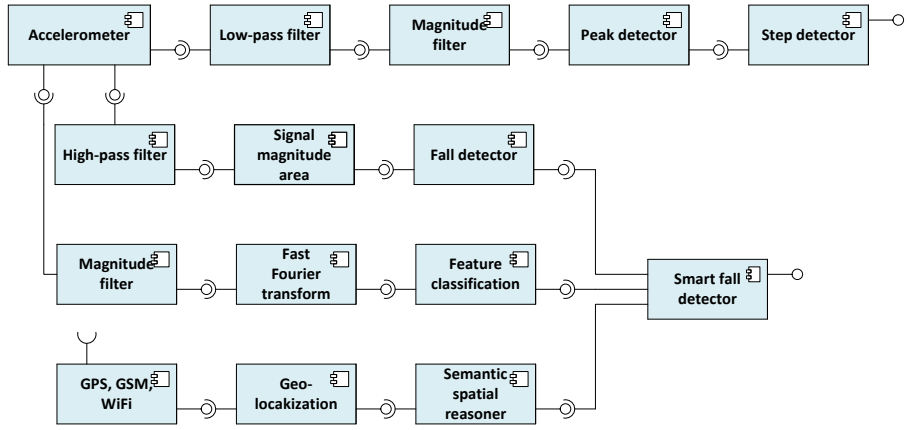


Figure 5.2: UML 2.0 component diagram for the fitness monitoring and fall detection application

Building blocks of the fitness monitoring and fall detection application

Figure 5.2 provides an overview of the composition of the components for the first use case. We will describe each of these components below.

- **Accelerometer**: This component produces a continuous stream of acceleration data by sampling a tri-axial accelerometer at a certain sampling rate.
- **Low-pass filter**: For mobility tracking, e.g. walking or running, we are mainly interested in acceleration peaks that arrive at a frequency of maximum 5Hz (i.e. max 5 steps per second). Therefore, a 'moving average' component is used as a simple low-pass filter to remove high-frequency noise.

- **Magnitude filter**: Often, we do not know how the accelerometer is oriented at the offset of the motion activity. Furthermore, the orientation of the sensor may change while moving around. Therefore, the signal analysis is done on the overall magnitude of the acceleration signal.

- **Peak filter**: A single step is characterized by a pattern of several maxima and minima in the time domain of the acceleration signal. This component extracts these features in the signal for further analysis.

- **Step detector**: Identifies the peaks for every step in order to correctly count the number of steps and to differentiate between standing still, walking and running (i.e. the peak rate).

- **High-pass filter**: For fall detection, we are interested in sudden and high-frequency changes of the acceleration signal, both in amplitude and orientation. This component implements a high-pass Finite Impulse Response (FIR) filter to extract these features.

- **Fall detector**: Analyzes the signal magnitude area (SMA) of the high-frequency part of the acceleration signal, and identifies a fall if this feature passes a certain threshold.

- **Fast Fourier Transformation**: Takes a time domain signal and converts it into a frequency domain signal. It is used for feature extraction on the magnitude signal and provides input for activity training and classification.

- **Feature classification**: Whereas the previous components mainly condition and extract useful features from the accelerometer data, this component builds decision models through training (leveraging the Weka machine learning library) in order to classify the activity of the user.

- **Geo-localization unit**: This component uses a variety of algorithms and filters to compute in-door localization, based on signal strength and time of arrival from a range of nodes, in order to localize the users and associated objects in real-time.

- **Semantic spatial reasoner**: A parliament based semantic reasoner is used to map the geo-location information from the previous component to meaningful semantic descriptions (e.g., kitchen, couch in the living room, etc) that are better understandable for the end users.
- **Smart fall detector**: A component that co-relates the falls and the semantic location of the user. It is a probabilistic model that calculates the risk and analyzes the emergency situation to notify the caregiver.

This system utilizes the built-in tri-axial accelerometer to do opportunistic sensing. Although the energy efficiency of the accelerometers has increased over the years, continuous sensing at high frequencies and on-board processing of these data streams have proven to drain the battery of mobile devices rapidly [87]. Hence, a major challenge with sensing is to determine the optimal sampling frequency and extracting the suitable features depending on the other available context information. Despite using opportunistic sensing, precise labeling of the training data is a significant challenge, as most of the models require extensive training data for good classification accuracy. This problem worsens as the scale of this system increases. Therefore, the primary motivation is to build a scalable activity recognition system for mobile devices with intelligent hybrid sensing, inference and learning that can leverage the resource rich environment of the cloud.

5.3 Overview of the deployment and configuration methodology

It is impossible to determine in advance where each software component will run due to the dynamic interaction of these devices with the environment and the user. The variety of parameters associated with the several possible configurations under varying workload and resource availability makes it almost impossible to manually fine-tune the components for best overall system performance, necessitating automated deployment, configuration and adaptation. Furthermore, performing a

detailed cost benefit analysis for configuration and adaption decisions from scratch at runtime causes a large overhead. However, this overhead can be reduced by balancing the offline and runtime efforts of making dynamic deployment decisions.

Figure 5.3 gives an overview of our approach which consists of two phases: a design-time phase and a runtime phase. At design time, application components are profiled and optimal configurations for a set of possible runtime situations are computed along with the costs of switching between these configurations. At runtime, these pre-computed configurations are used as per the context of the user and other runtime parameters of the system.

5.3.1 Design time phase

Figure 5.4 gives an overview of the offline exploration for the pre-processing of deployment and configuration decisions. The component-based application is first profiled to obtain an annotated component graph. This annotated component graph is used for the exploration of the Pareto-optimal deployments and configurations. A reconfiguration cost matrix is constructed only for these Pareto-optimal configurations. The runtime system uses the explored Pareto-optimal configurations and the reconfiguration matrices in order to make self-optimization decisions at runtime.

Profiling

We use annotated component graphs, a high-level model of computation to represent the application in order to explore the trade-offs between different deployment configurations of the application. An annotated component graph is a directed graph where the nodes represent the components of an application, and the edges represent the dataflow between the components. These nodes and edges are annotated with meta-data representing the hard constraints, costs and resource requirements of the components, i.e. the CPU time, energy consumption and memory requirements for the components and the amount of data being transferred along the edge. This meta-data is collected by profiling

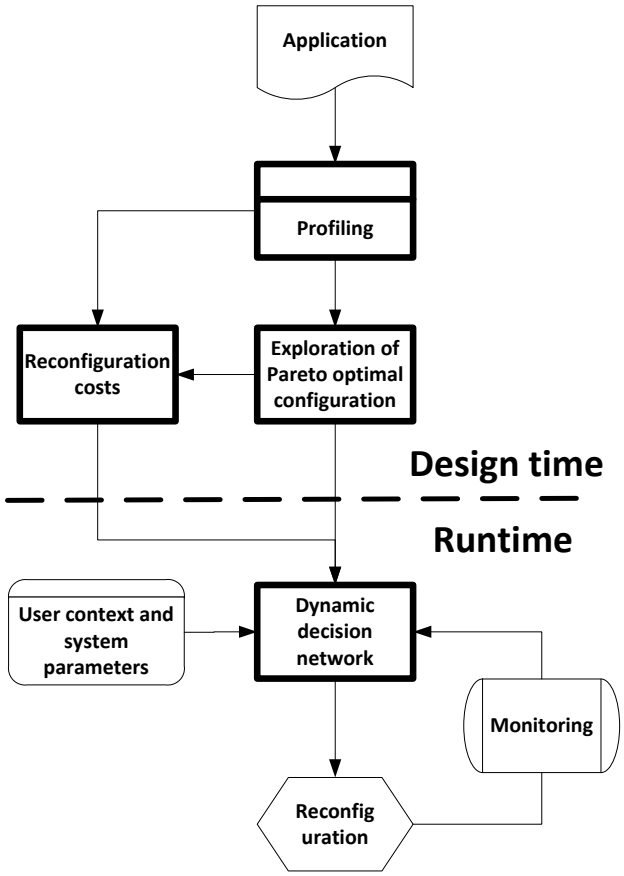


Figure 5.3: Overview of the approach illustrating the offline and runtime phases

the execution of components on different platforms. In ubiquitous computing these different platforms are broadly ranged; therefore, we use the classification of *Smart Objects*, *Smart Mobiles* and *Smart Servers*:

- *Smart Object*: Small appliances, sensors or actuators with limited computational power, storage capacity, communication capability, energy supply and primitive user interface are categorized as smart objects (e.g. RFID tagged objects, motion detectors, heating regulators).

- *Smart Mobile*: Devices with multi-modal user interfaces to enable user mobility through remote services are categorized as smart mobiles (e.g. smart phones, smart TVs). They usually have better resource provisions than smart objects.
- *Smart Server*: The aggregation and complex analysis of data from smart objects and smart mobiles are realized as services on smart servers (e.g. a local server or remote cloud computing set-up).

In order to generate an annotated component graph for this application, following steps are carried out:

1. Use the component model of the application and identify the dataflow (similar to the one shown in Figure 5.2) between them. The dataflow graph acts as a skeleton for the annotated component graph.
2. Instrument the communication interfaces of components to measure the amount of data transferred between them.
3. Run every component of the application on all the different platforms possible, profiling its execution time, energy consumption and data transferred between components, each time.
4. Calculate the memory requirements of every component by monitoring the changes in stack and heap sizes, as components are added and removed from the platform.
5. Repeat step 3 and 4 over a range of component configurations (e.g. a different sampling rate) and/or simulated inputs (e.g. accelerometer traces of different activities and individuals).

The hard constraints for the second use case are that the accelerometer and the GPS can only execute on devices with such sensors. Some components have configuration options that affect their resource costs and requirements. For example, lowering the accelerometer sampling rate from 50Hz to 15Hz decreases the CPU time, communication and energy consumption of the activity recognition components, but increases the

recognition error rate. For such components we annotate the component graph with meta-data for a discretized range of parameter options, i.e. the CPU time and energy consumption values for the supported sampling rates. Adding all this meta-data to the dataflow graph generates the annotated component graph of the application and we use it as an intermediate model for exploring Pareto-optimal deployment trade-offs at design time.

Pareto exploration

We model the problem of deploying an application to a heterogeneous network of (re)configurable nodes as a constraint-based optimization problem and use a CPLEX based solver to explore the Pareto-optimal set of solutions. In a Pareto-optimal set of solutions, every solution is better than all other solutions according to at least one functional or non-functional criterion.

Eliminating deployment and configuration options that are not Pareto-optimal reduces the search space for the runtime reconfiguration decision, from all possible configurations to only the set of Pareto-optimal configurations. For example, consider the use case in section 3 which consists of 13 components, 11 of these components may be deployed on three different platforms. Deploying more components on the *Smart Objects* and *Smarts Mobiles* stresses these devices in terms of memory, computational power and battery consumption, whereas, deploying more components on the *Smart Server* adds to the communication costs. Moreover, some of the components have different operational modes (e.g. the sampling frequency for the accelerometer component) with different costs. All of these permutations when combined form a large solution space and exploring it at runtime would incur a huge overhead. Therefore, at design time a set of Pareto-optimal configurations are computed (in order to reduce the solution space of the runtime phase).

In order to explore multi-dimensional Pareto-optimal surfaces, the problem is modeled using parameterizable constraints. These parameters are then iteratively varied over a discretized range, invoking the solver each time to find a point on a Pareto surface. For example, an energy consumption versus Quality of Service Pareto curve is explored for the

step counting algorithm by iteratively finding minimum energy solutions for different QoS constraints. It is important to note that there are no dependencies among the different invocations of the CPLEX solver. While finding solutions for this application takes several minutes on a single machine (depending on how many simulations are carried out), we can speed up this process by initiating parallel invocations of the CPLEX solver on a cluster of machines. This guarantees the feasibility of the approach for larger applications with many more configuration alternatives.

Exploring reconfiguration costs

The Pareto front provides configurations that are optimal for a given runtime situation. However, these configurations are not feasible or the most optimal for all runtime situations. Therefore, switching from one Pareto-optimal configuration to another is often required. Let us hypothetically consider an application with two runtime situations: *context A* and *context B*, and two Pareto-optimal configurations: *configuration X* and *configuration Y*. Configuration X is cheaper than configuration Y. However, it is only valid for context A and switching between the configurations requires N mAh. Now if the system is in the configuration Y and the context changes from B to A, the runtime system has two options either to stay in configuration Y or to switch to the cheaper configuration X. The decision whether to switch to configuration X or to stay in configuration Y depends on how long the context A will last. If it lasts only for a very short period, the cost of switching is not recovered. The final decision whether to switch between configurations or not is done at runtime, however, a table representing the costs of switching between different configurations is compiled at design time.

Some components have stochastic non-functional performance properties. For example, the communication throughput of a wireless node could be affected by external factors (e.g. interference). To define the Pareto-fronts (or Pareto-curves) for closed real-time systems the worst case execution estimates are usually taken after profiling. Given that the IoT ecosystem is quite heterogeneous and open ended in nature, pursuing such a pessimistic approach will easily lead to undesirable solutions.

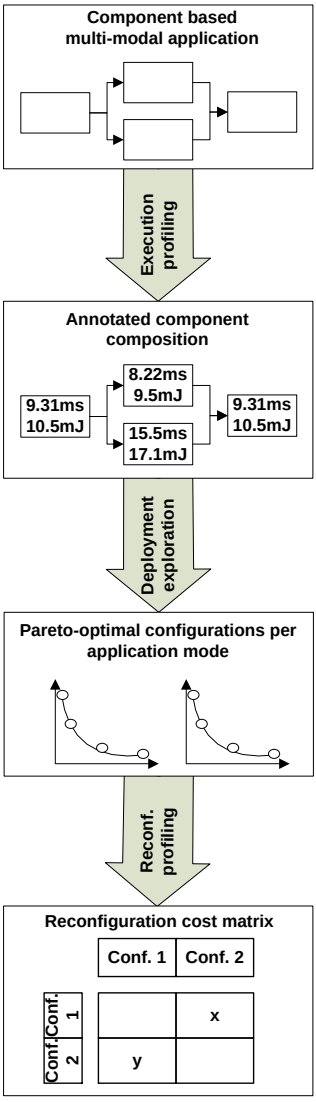


Figure 5.4: Overview of the offline exploration phase

We define the Pareto-points based on the most likely execution values. However, to be able to assess the impact of a worst case execution scenario for a particular deployment and configuration (i.e. a specific

Pareto-point), we incorporate the likelihood distribution of the profiled execution values in each Pareto-point leading to a Pareto-front (i.e. a set of Pareto-optimal solutions) with some degree of variability. A reconfiguration cost matrix is constructed by profiling the costs of reconfigurations and redeployment of components. For example, the cost of activation/deactivation of a component, establishing a local/remote component-to-component communication channel and transferring the state of an active component over a communication network. The size of this matrix is $O(N^2)$ where N is the number of possible configurations. As N can become large, only the Pareto-optimal configurations are considered for reconfigurations.

5.4 Results and discussions

This section provides the results and discussion related to applying the methodology presented in section 5.3 on the applications presented in section 5.2.

5.4.1 Energy-aware application deployment on CaPI based WSNs

In order to explore the deployment configurations of the use case presented in section 5.2.1, the same functions were implemented as components and as policies on both Ravens and SunSpots. The minimum energy configurations are explored for varying number of messages sent for the four different representative scenarios. Two aspects of the software artifacts (components and policies) are profiled on both platforms (Raven and SunSpot), i.e. deployment overhead and execution overhead. The overheads are profiled in terms of time. The energy dissipation is calculated using power models based on the power values provided in the respective platform datasheets.

Figure 5.5 shows the exploration results for the selection of the optimal artifact in a scenario where one Raven node is sending messages to another Raven node. We observe that deploying a conversion policy is the optimal choice for this scenario if less than 45 messages are sent,

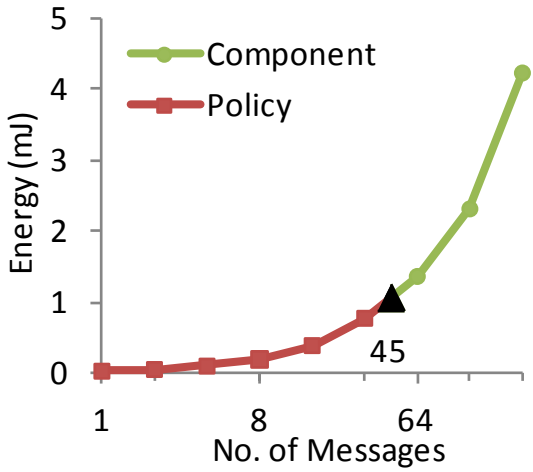


Figure 5.5: One Raven node sending messages to another Raven node

however, if more than 45 messages are sent then deploying a conversion component becomes the optimal choice. This is because policies on Ravens are smaller and therefore cheaper to deploy but less efficient, whereas components are larger and therefore more expensive to deploy but more efficient. Figure 5.6 shows a similar exploration results for a scenario where ten Raven nodes are sending messages to a Raven cluster head. In all the solutions a policy or a component is deployed only on the cluster head. In this case the critical point after which a component is more energy efficient than a policy is at five messages.

Figures 5.7 and 5.8 show exploration results for scenarios where a Sun SPOT cluster head receives messages from ten and hundred Raven nodes respectively. For both these scenarios, we see that for a very small number of messages (the solution with the least setup cost) deploying a policy only on the Sun SPOT cluster head is the optimal solution. For a slightly larger number of messages, deploying policies and components on all the Raven nodes is the optimal choice, as the Raven nodes consume lesser power compared to the Sun SPOT node. It is also noted that the option of deploying a component on the Sun SPOT is never chosen. Since the components on SunSPOTs are larger they have more performance overhead compared to policies.

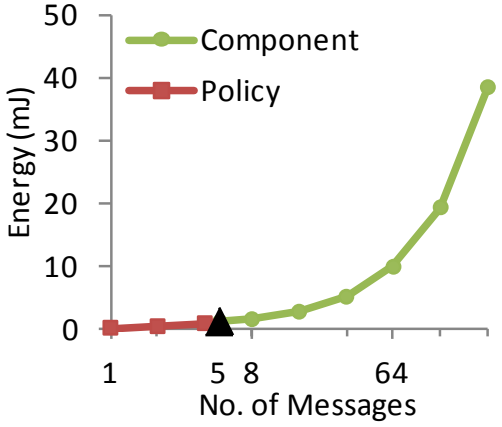


Figure 5.6: Ten Raven nodes sending messages to a Raven cluster head

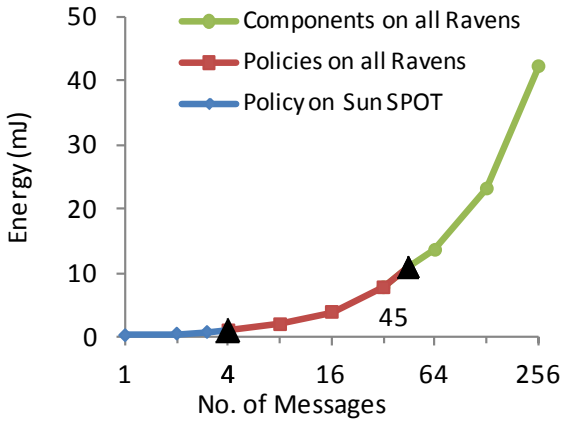


Figure 5.7: Ten Raven nodes sending messages to a Sun SPOT cluster head

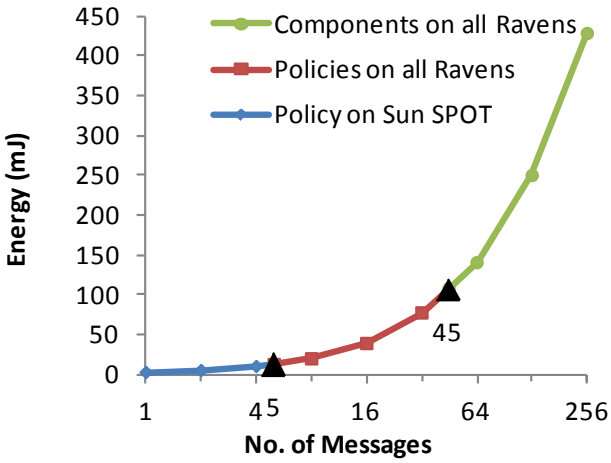


Figure 5.8: Hundred Raven nodes sending messages to a Sun Spot cluster head

5.4.2 Deployment trade-offs of fitness monitoring and fall detection application

This section presents results of our second use case. First, profiling results of some components are presented in table 5.1 and then two important aspects of this use case are discussed; the exploration of functional and non-functional trade-offs and second the variability of non-functional properties.

Profiling

The profiling results of the step counting application on the SunSPOT sensor are shown in Table 5.1. Note that for the *Accelerometer, Low-pass filter* and *Magnitude filter* components, there is little to no communication variability because the amount of data output is fixed and depends on the sampling rate of the accelerometer. We were not able to test all the components on the Raven sensor, but those that were ported executed approximately 50 times slower compared to the SunSPOT.

Component	CPU load	Communication
Accelerometer	8.09 ± 1.3 ms	5.5 ± 0.0 kB/sec
Low-pass filter	57.9 ± 2.1 ms	5.5 ± 0.0 kB/sec
Magnitude filter	18.2 ± 1.5 ms	1.8 ± 0.0 kB/sec
Peak detector	14.9 ± 9.7 ms	0.5 ± 0.4 kB/sec
Step detector	5.12 ± 4.8 ms	0.1 ± 0.1 kB/sec
FFT	5590 ± 108 ms	2.5 ± 0.0 kB/sec
High-pass filter	197 ± 8.2 ms	5.0 ± 0.0 kB/sec
Signal Magnitude Area	51.7 ± 3.4 ms	2.0 ± 0.0 kB/sec
Fall detector	15.1 ± 9.1 ms	2.0 ± 0.1 kB/sec

Table 5.1: Performance benchmark of the individual components on the SunSPOT sensor

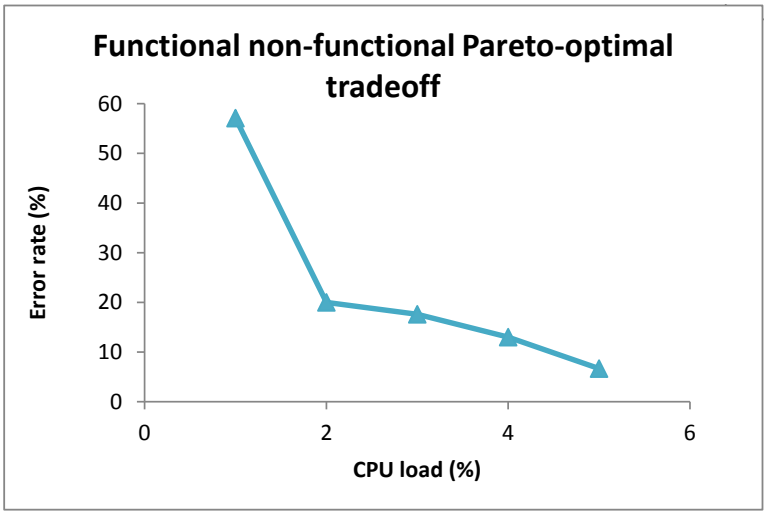


Figure 5.9: Functional non-functional Pareto-optimal trade-offs between quality of service (error rate) and CPU load on the SunSPOT.

Exploring functional/non-functional trade-offs

Ubiquitous computing applications often need to make trade-offs between functional and non-functional optimization objectives, e.g., trade-offs between quality of service and the cost of it. Figure 5.9 depicts one such

trade-off for step counting. The error rate provides a measure for the quality of service and the CPU load represents its cost. This trade-off depends on the sampling frequency of the accelerometer component. Higher sampling frequency results in a smaller error rate but at the cost of increased CPU load. For 24/7 mobility monitoring it is not desirable to have a static sampling frequency, for example, the sampling frequency may be lowered in order to save energy if the user is sleeping and increased if he/she is walking or running. Dynamic decision networks [98] can use this exploration data to adjust the sampling frequency at runtime.

Outlook on managing variability

In the second use case we observe two types of dynamism:

1. The first form of dynamism is reactive, i.e. changing execution modes or parameters of the application in reaction to changes in the context; e.g., changing the sampling frequency of the accelerometer in different scenarios (sleeping, walking, running etc.). This form of dynamism is manageable with scenario-based or context-aware methodologies.

2. The second form of dynamism is low level, i.e. the variability within a scenario. For example, the computational load of the Step Detector component (see Figure 5.2) is different for different users because people have different walking patterns. This form of dynamism is not manageable with scenario-based or context-aware methodologies at design time.

By learning user specific patterns it is possible to predict a part of this variability, hence optimizing deployment and configuration. Therefore, we propose using a fuzzy Pareto space for such applications. A fuzzy Pareto curve takes into account the variability of non-functional properties of different components when exploring the deployment configurations. An intelligent runtime system learns user patterns and uses them to fine tune and further optimize the configurations at runtime [106].

Figure 5.10 illustrates a fuzzy Pareto space of the trade-off between CPU load and network communication for different deployment configurations. Given the fact that the deployment of the *Accelerometer* component is fixed, we have 16 different deployment options for the 4 remaining components. Figure 5.10 shows the five Pareto-optimal ones, the average trade-off point is marked by the red dots and the variability is shown as circles around these red dots. Note that we assume perfect normal distributions here, in real-life the variability is likely to be skewed in one or more directions.

- D_1: Minimal computation on the sensor by having the *Accelerometer* component on the sensor and the 4 remaining sensor data processing components deployed on the server.
- D_2: The *Accelerometer* and *Low-pass filter* components deployed on the sensor and the other components on the server.
- D_3: The *Accelerometer*, *Low-pass filter* and *Magnitude filter* components deployed on the sensor and the other components on the server.
- D_4: All components except the *Step Detector* component deployed on the sensor.
- D_5: Highest CPU load on the sensor by having all the components deployed on the sensor and no communication cost between the sensor and the server.

Note that each deployment D_x represents the joint resource consumption and variability of the components deployed on the sensor. Examples of non-Pareto-optimal solutions include a.o. a deployment with the *Low-Pass Filter* and *Peak Detector* components on the server and the *Accelerometer*, *Magnitude Filter* and *Step Detector* components on the sensor. This mixed deployment causes a high communication cost.

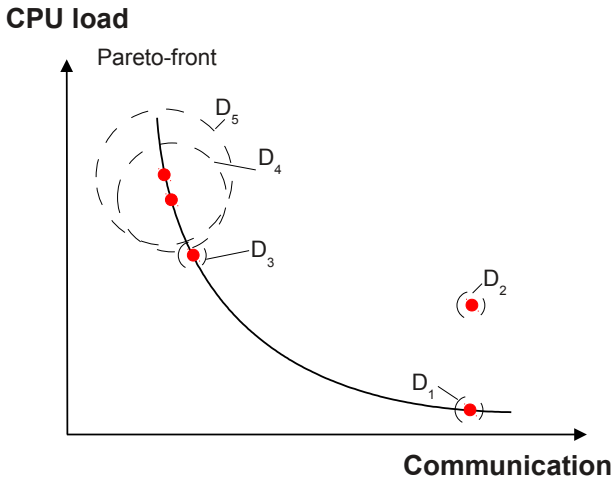


Figure 5.10: A fuzzy Pareto space of the tradeoff between CPU load and network communication for different deployment configurations

5.5 Conclusions

This chapter applies a dataflow-based modeling and mapping techniques for the deployment and configuration of ubiquitous computing applications. Results from two use cases are presented: (1) A component and policy infrastructure based WSN and (2) Mobility monitoring and fall detection application. The first use case results demonstrate the ability of the approach to find optimal deployment configurations for different scenarios and topologies in a distributed and heterogeneous system. The second use case takes a step further by relaxing the fixed quality of service constraints and exploring the trade-offs between quality of service and CPU load. A major drawback of this approach is its lack or ability to deal with low-level fine-grained variability, such as random jitters in senor data or network variability. We present an improvement to the approach in the form of fuzzy Pareto space in order to deal with this limitation.

Chapter 6

Conclusion and future work

This chapter concludes the research presented in the thesis and proposes directions for future research. Section 6.1 summarizes the contributions of the thesis. Section 6.2 highlights some important lessons learned from the research. Section 6.3 presents some critical reflections and section 6.4 proposes directions for future work.

6.1 Recapitulating the contributions

In this thesis we present dataflow-inspired techniques for the development and execution of parallel and distributed programs. We apply these techniques in real world use cases and validate them. This section summarizes the major contributions of the dissertation.

Contribution 1: Parallel stencil operation using dynamic task graphs

The first contribution of this thesis is the modeling and programming of a kernel commonly used in high performance computing applications, using a dataflow centric approach. Chapter 3 presents an implementation of parallel stencil operations using dynamic task graphs. Two benchmark applications are used for the evaluation of the approach and the results are compared to the state-of-the-art stencil compilers Pluto [15] and

Pochoir [126]. Avoiding global barriers and using point-to-point non-blocking synchronization mechanisms results in a reduction in spin-waiting times of the processors. Reduction in total synchronization overhead by factors of fifty and five compared to Pluto and Pochoir respectively, are recorded on a forty-core machine. Furthermore, the results show that the dynamic task graph based implementation scales better than the state-of-the-art.

Contribution 2: Design-time exploration of Pareto optimal mappings for embedded applications

The second contribution is the improvement of scheduling and memory management of data programs. Chapter 4 presents a design-time co-exploration technique that schedules tasks on processors, data objects on the memories and data transfers on the interconnect. The inter-dependencies between the three different types of schedules are incorporated in the exploration models. Two use cases are utilized to study the effects of horizontal and vertical coupling. Horizontal coupling is realized by modeling the inter-dependencies between different aspects of mapping i.e. memory, communication and computation. Vertical coupling is realized by modeling inter-dependencies within a domain, e.g., the inter-dependencies between buffer sizes and scratchpad allocations. Our results for the two real-life application use-cases show significant (at least 20%) improvement with horizontal or vertical coupling.

Contribution 3: Dataflow centric component deployment for ubiquitous computing applications

The third contribution is applying a dataflow based methodology for the deployment and configuration of component based applications in ubiquitous environments. Exploration results for two use cases are presented: (1) Component and policy infrastructure based WSN and (2) mobility monitoring and fall detection application. The first use case demonstrates the ability of the approach to find optimal deployment configurations for different scenarios and topologies in a distributed and heterogeneous system. The second use case explores the functional/non-

functional tradeoff between quality-of-service and CPU load for different deployment configurations.

6.2 Important lessons learned

This section summarizes some important lessons learned during the course of the research for this dissertation.

Dataflow models as enablers for efficient parallel programming

The benchmarks and use-cases studied in this thesis show that dataflow based programming models and languages provide efficient solutions for programming, memory management, scheduling and synchronization of parallel programs. This success is attributed to some basic properties of the model of computation:

- Implicit parallelism: Programmers only specify the functionality of the application and are not required to explicitly specify parallelism. Parallelism is an intrinsic property of the programming model.
- Locality of effects allow a race condition free execution of dataflow graphs, regardless of the choice of the scheduling policy.
- Schedulability and memory buffer size analysis at design time simplifies execution of dataflow models.
- Data-driven coordination of execution: The coordination between concurrent parts of a program is through the flow of data and the programmer is not burdened with the specification of explicit synchronization mechanisms.

Essentiality of domain knowledge for optimization

We observe that domain knowledge is essential in order to exploit available optimization opportunities using dataflow models. Two examples from contributions 1 & 2 are discussed below:

- Dynamic task graphs are used to program stencil operations, hence the synchronization overhead is reduced. However, only reducing the synchronization overhead of stencil operation does not result in a very significant speed up because the algorithm is bottle-necked by memory bandwidth. Domain knowledge is required to identify and address this bottleneck, i.e. through time-tiling. A significant speed-up by using dynamic task graphs is only observed once the memory bandwidth bottleneck is removed.
- We use a dataflow model for mapping a legacy H.264 decoder application on an MPSoC. However, deep understanding of the application domain is required in order to partition the application and derive a dataflow model. The quality of the mapping solutions depends on the partitioning of the application; a badly partitioned application cannot be efficiently mapped. Domain knowledge is thus essential for the efficient mapping of the application.

Conservative use of worst-case estimates

For highly dynamic applications, the variability of non-functional properties is sometimes too high for a given deployment configuration, prohibiting worst-case estimates. For example, in the mobility and fitness monitoring application use case in chapter 5, the variability of some components is up to $\pm 80\%$ from the average for the communication (in kB/s) and $\pm 94\%$ from the average for computation (ms of CPU time). Therefore, for non-safety critical soft-real time applications, we propose a fuzzy Pareto curve based methodology instead of worst-case estimates.

6.3 Critical reflections

One of the main goals of this thesis is to strengthen the case for dataflow models as generic solutions for parallel and distributed computing. A lot more is required to fully achieve this goal. Not only the methodologies must be applied in a number of other sub-domains of parallel and distributed computing but also more use cases in each of these sub

domains (i.e. high performance computing, embedded systems and ubiquitous systems) must be evaluated.

According to the execution semantics of pure dataflow models, processes or tasks go through explicit read-execute-write sequences. On a Von Neumann computer, strict adherence to these semantics creates unnecessary traffic, especially on a shared memory computer. We circumvent this problem by letting processors communicate with each other by passing address pointers, if the data is located in a physically shared memory. As a result, our implementations do not strictly comply with execution semantics of pure dataflow models.

The following sub-sections provide some critical reflections structured according to the contributions.

6.3.1 Parallel stencil operations using dynamic task graphs

Dimensions and order of stencil operations

Chapter 3 presents results for two benchmarks based on two-dimensional 5pt. and 9pt. stencil operations. However, many high-performance computing applications use higher order stencil operations possibly with more dimensions. These applications may have different performance bottlenecks, e.g., a tri-cubic interpolation has a FLOPS/byte ratio of 7.95 and the bottleneck is computation rather than memory bandwidth or latency. Even though using dynamic task graphs would reduce the synchronization overhead for these stencil operations as well, a significant performance improvement might not be achieved unless other bottlenecks are removed.

Static and regular grids

The benchmarks used for stencil operations in chapter 3 have regular and static grids. However, some high-performance computing applications use stencil operations on grids with irregular resolutions, e.g., higher resolutions in regions of specific interest. Other applications adapt the resolution of different regions at runtime. Some techniques presented in

chapter 3 (e.g. time-tiling and memory de-allocation scheme) are not directly applicable in these situations.

Distribution across multiple nodes

HPC applications often execute while distributed over networks of multiple nodes. The tradeoffs of this distribution (e.g. ratio of computational power versus interconnect bandwidth) are very different. Our technique improves the deployment and execution of stencil operations on parallel processors within a single node. Even though a similar technique may be used for deployment of the application across multiple nodes, it is not been experimentally evaluated.

6.3.2 Design-time exploration of Pareto optimal mappings for embedded applications

Increase in the computational complexity

Chapter 4 presents tightly coupled mapping of embedded applications on MPSoC. However, tightly coupled mapping requires that the inter-dependencies between computation, communication and memory are incorporated in the exploration models. A side effect of including these inter-dependencies in the exploration models is an increase in the computational complexity, thus a decrease in the speed of exploration.

Evaluation of the runtime manager

A runtime manager is required to orchestrate, processing, data and communication on the MPSoC. The overall performance of the system depends on the quality of the mapping solutions and the overhead of the runtime manager. In this thesis, only the quality of the mapping solutions is evaluated and the overhead of the runtime manager is not evaluated.

6.3.3 Dataflow centric component deployment for ubiquitous computing applications

Closed world assumptions

The methodology presented in chapter 5 relies on closed world assumptions. Most real-life ubiquitous computing systems are open-ended, i.e., devices join and exit the system on the fly. Even though it is theoretically possible to model this behavior with scenarios, it causes an explosion in the number of scenarios that have to be explored. Therefore, more proactive and dynamic runtime solutions may be required to deal with these situations.

6.4 Future work

This section discusses directions for future research.

6.4.1 Dynamic task graph based stencil operations for applications with adaptive mesh refinement

Adaptive mesh refinement (AMR) is employed in a number of scientific and engineering applications that use stencil operations. In adaptive mesh refinement [21] the resolution of different areas of the grid may change at runtime. This complicates domain partitioning and workload distribution for parallelized implementations. Furthermore, usual optimization techniques (e.g. time-tiling) for stencil operations are not directly applicable and further research is required to apply them. Although it is a non-trivial extension, the ability of dynamic task graphs to express dynamism makes them suitable for programming AMR based applications. Therefore, it is promising to investigate the applicability of dynamic task graphs for AMR applications.

6.4.2 Mapping embedded real-time applications on network-on-chips

Network-on-Chip (NoC) is an evolution of the multi-core system on chip. In a NoC, modules such as processors, memories and other specialized blocks exchange data using a 'network' sub-system. The network sub-systems usually consist of numerous point-to-point data links and switches (router). Unlike a simple bus-based system, the communication over this network is usually multi-hop. By including network communication models in the co-exploration technique presented in chapter 4, it is possible to extend it for mapping applications to Networks-on-Chips.

Glossary and acronyms

Base station
: a fixed communications infrastructure that is part of a wireless network. It relays information to and from wireless nodes enabling them to work within a local area .

Component based application
: Software applications developed by composing software modules (*components*) that package together related functions and have defined interfaces.

Constraint based scheduling
: A technique where scheduling is defined as an optimization problem such that a set of constraints have to be satisfied.

Core
: Refers to a single processor typically in multi-processor system.

Deployment
: Distribution of artifacts in a distributed systems.

Discretization
: The process of transferring continuous models and equations into discrete counterparts.

Distributed system
: a software system in which components located on networked computers communicate and coordinate their actions by passing messages .

Domain
The set of input or argument values for which the function is defined.

DSP
Digital signal processor.

Energy harvesting
A process by which energy is derived from the environment.

GPU
Graphics processing unit.

H.264
A video compression format.

Heterogeneous cores
CPU cores with different characteristics, e.g., having different architecture, frequency, computational power or energy efficiency.

Hyper-trapezoid
A trapezoid extended in the time domain.

Interconnect
Communication infrastructure for example network or bus.

Mapping
Process of deployment and scheduling software applications on hardware platforms.

Memory hierarchy
A hierarchically organized memory sub-system, usually with smaller and faster memories close to the processors.

Microcontroller
A small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals .

MPSoC
Multi-processor system on chip.

Nomadic
Pertaining to the characteristic of nomads, e.g. mobility.

Non-affine
A function that can not be expressed in the form of $f(\vec{x}) = A\vec{x} + B$.

Open-ended
Not restrained by definite limits, restrictions, or structure.

Pareto optimal solutions
A set of solutions such that every solution in the set is better from all other solutions with respect to at least one criterion.

Power mode
A mode of operation with specific electrical characteristics, typically modes that deliver higher performance require more input power.

Real-time systems
Systems that must guarantee response within strict time constraints (deadlines).

Reconfiguration
To change the arrangement of components or elements in a particular form, figure, or combination.

RFID
Radio-frequency identification (RFID) is the wireless non-contact use of radio-frequency electromagnetic fields to transfer data, for the purposes of automatically identifying and tracking tags attached to objects.

RISC
Reduced instructions set computing.

Scratchpad
Software managed small and fast memory, similar to a cache in

latency and bandwidth but without a dedicated hardware cache controller.

SIMD
Single instruction multiple data.

Spatio-temporal schedule
A plan of when and where (time and space) certain activities will take place.

Spin-waiting
A technique in which a process repeatedly checks to see if a condition is true, e.g., if lock is available or not.

Stochastic
having a random probability distribution or pattern that may be analyzed statistically but may not be predicted precisely.

Streaming applications
Applications that stream data from a server; e.g, audio/video streaming.

Throughput
A measure of the flow of data, e.g. through a network of nodes or a pipeline of processes.

Timestep
A small increment in time for example when simulating a physical phenomonon.

Trapezoid
A convex quadrilateral with at least one pair of parallel sides.

Ubiquitous computing (ubicomp)
an advanced computing concept where computing is made to appear everywhere and anywhere .

Virtual machine
A software-based emulation of a computer, that executes programs like a physical machine but limited to the resources and abstractions provided by the virtual machine.

VLIW
Very long instruction word.

Wireless sensor network (WSN)
A spatially distributed network of autonomous sensors; for example to monitor physical or environmental conditions, such as temperature, sound, pressure, etc. and to cooperatively pass their data through the network to a main location.

Bibliography

[1] R. K. Abbott and H. Garcia-Molina. "Scheduling Real-time Transactions: A Performance Evaluation". In: *ACM Trans. Database Syst.* 17.3 (Sept. 1992), pp. 513–560. ISSN: 0362-5915. DOI: 10.1145/132271.132276. URL: http://doi.acm.org/10.1145/132271.132276.

[2] M. Ade, R. Lauwereins, and J. Peperstraete. "Data Memory Minimisation For Synchronous Data Flow Graphs Emulated On DSP-FPGA Targets". In: *Design Automation Conference, 1997. Proceedings of the 34th.* 1997, pp. 64 –69. DOI: 10.1109/DAC.1997.597118.

[3] G. A. Agha. "Actors: A Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence)". AAI8520855. PhD thesis. Ann Arbor, MI, USA, 1985.

[4] K. Agrawal, C. Leiserson, and J. Sukha. "Executing task graphs using work-stealing". In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* 2010, pp. 1 –12. DOI: 10.1109/IPDPS.2010.5470403.

[5] F. Angiolini, L. Benini, and A. Caprara. "An efficient profile-based algorithm for scratchpad memory partitioning". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 24.11 (2005), pp. 1660–1676. ISSN: 0278-0070. DOI: 10.1109/TCAD.2005.852299.

[6] M. Idrissi Aouad and O. Zendra. "A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy".

In: *2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*. ECOOP. Berlin Allemagne, 2007, pp. 31–38.

[7] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1996.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198. ISSN: 1532-0634. DOI: 10.1002/cpe.1631. URL: http://dx.doi.org/10.1002/cpe.1631.

[9] O. Avissar, R. Barua, and D. Stewart. "An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 1.1 (Nov. 2002), pp. 6–26. ISSN: 1539-9087. DOI: 10.1145/581888.581891. URL: http://doi.acm.org/10.1145/581888.581891.

[10] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. "The Design of OpenMP Tasks". In: *Parallel and Distributed Systems, IEEE Transactions on* 20.3 (2009), pp. 404–418. ISSN: 1045-9219. DOI: 10.1109/TPDS.2008.105.

[11] T. P. Baker. "An Analysis of EDF Schedulability on a Multiprocessor". In: *IEEE Transactions on Parallel and Distributed Systems* (2005). ISSN: 1045-9219. DOI: 10.1109/TPDS.2005.88. URL: http://doi.ieeecomputersociety.org/10.1109/TPDS.2005.88.

[12] N. Bambha and S. Bhattacharyya. "Communication strategies for shared-bus embedded multiprocessors". In: 2005.

[13] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems". In: Estes Park, Colorado, USA, 2002, pp. 73–77.

[14] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. "Scratchpad memory: design alternative for cache on-chip memory in embedded systems". In: *Proceedings of the tenth international symposium on Hardware/software codesign*. CODES

'02. Estes Park, Colorado: ACM, 2002, pp. 73–78. ISBN: 1-58113-542-4. DOI: http://doi.acm.org/10.1145/774789.774805. URL: http://doi.acm.org/10.1145/774789.774805.

[15] V. Bandishti, I. Pananilath, and U. Bondhugula. "Tiling stencil computations to maximize parallelism". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 40:1–40:11. ISBN: 978-1-4673-0804-5.

[16] S. Bandyopadhyay. "Automated Memory Allocation of Actor Code and Data Buffer in Heterochronous Dataflow Models to Scratchpad Memory". MA thesis. EECS Department, University of California, Berkeley, 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-105.html.

[17] S. Bandyopadhyay, T. H. Feng, H. D. Patel, and E. A. Lee. *A Scratchpad Memory Allocation Scheme for Dataflow Models*. Tech. rep. UCB/EECS-2008-104. EECS Department, University of California, Berkeley, 2008. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-104.html.

[18] S. K. Baruah. "Scheduling periodic tasks on uniform multiprocessors". In: *Inf. Process. Lett.* (2001).

[19] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. "A compiler framework for optimization of affine loop nests for gpgpus". In: *Proceedings of the 22nd annual international conference on Supercomputing*. ICS '08. Island of Kos, Greece: ACM, 2008, pp. 225–234. ISBN: 978-1-60558-158-3. DOI: 10.1145/1375527.1375562. URL: http://doi.acm.org/10.1145/1375527.1375562.

[20] L. Bathen, N. Dutt, and S. Pasricha. "A framework for memory-aware multimedia application mapping on chip-multiprocessors". In: *ESTImedia 2008*. 2008.

[21] M. J. Berger and J. Oliger. "Adaptive mesh refinement for hyperbolic partial differential equations". In: *Journal of Computational Physics* 53.3 (1984), pp. 484 –512. ISSN: 0021-9991. DOI: 10.1016/0021-9991(84)90073-1.

[22] B. Bhattacharya and S. Bhattacharyya. “Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems”. In: *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on.* 2000, pp. 84–89. DOI: 10.1109/IWRSP.2000.855200.

[23] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. “OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems”. In: *SIGARCH Comput. Archit. News* 36 (5 2009), pp. 29–35. ISSN: 0163-5964. DOI: http://doi.acm.org/10.1145/1556444.1556449. URL: http://doi.acm.org/10.1145/1556444.1556449.

[24] Z. Bhatti, D. Preuveneers, Y. Berbers, N. Miniskar, and R. Wuyts. “SAMOSA: Scratchpad aware mapping of streaming applications”. In: *System on Chip (SoC), 2011 International Symposium on.* 2011, pp. 48 –55. DOI: 10.1109/ISSOC.2011.6089687.

[25] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. “Cycle-static dataflow”. In: *Signal Processing, IEEE Transactions on* 44.2 (1996), pp. 397–408. ISSN: 1053-587X. DOI: 10.1109/78.485935.

[26] M. Bister, Y. Taeymans, and J. Cornelis. “Automatic segmentation of cardiac MR images”. In: *Computers in cardiology* (1989), pp. 215–218.

[27] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. “Cilk: an efficient multithreaded runtime system”. In: *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming.* PPOPP ’95. Santa Barbara, California, United States: ACM, 1995, pp. 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958.

[28] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. “An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms”. In: DATE ’10.

[29] S. Borkar. “Design challenges of technology scaling”. In: *Micro, IEEE* 19.4 (1999), pp. 23–29. ISSN: 0272-1732. DOI: 10.1109/40.782564.

[30] S. Borkar and A. A. Chien. “The Future of Microprocessors”. In: *Commun. ACM* 54.5 (May 2011), pp. 67–77. ISSN: 0001-0782. DOI: 10.1145/1941487.1941507. URL: http://doi.acm.org/10.1145/1941487.1941507.

[31] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. “Brook for GPUs: stream computing on graphics hardware”. In: *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*. Los Angeles, California: ACM, 2004, pp. 777–786. DOI: http://doi.acm.org/10.1145/1186562.1015800.

[32] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. “Readings in hardware/software co-design”. In: ed. by G. De Micheli, R. Ernst, and W. Wolf. Norwell, MA, USA: Kluwer Academic Publishers, 2002. Chap. Ptolemy: a framework for simulating and prototyping heterogeneous systems, pp. 527–543. ISBN: 1-55860-702-1. URL: http://dl.acm.org/citation.cfm?id=567003.567050.

[33] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[34] W. Che and K. S. Chatha. “Scheduling of Synchronous Data Flow Models Onto Scratchpad Memory-based Embedded Processors”. In: *ACM Trans. Embed. Comput. Syst.* 13.1s (Dec. 2013), 30:1–30:25. ISSN: 1539-9087. DOI: 10.1145/2536747.2536752. URL: http://doi.acm.org/10.1145/2536747.2536752.

[35] W. Che and K. Chatha. “Scheduling of synchronous data flow models on scratchpad memory based embedded processors”. In: *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*. 2010, pp. 205 –212. DOI: 10.1109/ICCAD.2010.5654150.

[36] J.-J. Chen and C.-S. Shih. “Energy-efficient Real-time Task Scheduling in Multiprocessor DVS Systems”. In: *ASP-DAC*. Yokohama, Japan, 2007, pp. 342–349. DOI: 10.1109/ASPDAC.2007.358009. URL: http://dx.doi.org/10.1109/ASPDAC.2007.358009.

[37] M. Christen, O. Schenk, and H. Burkhart. “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures”. In: *Parallel*

Distributed Processing Symposium (IPDPS), 2011 IEEE International. 2011, pp. 676 –687. DOI: 10.1109/IPDPS.2011.70.

[38] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. "Parallel data-locality aware stencil computations on modern micro-architectures". In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009, pp. 1 –10. DOI: 10.1109/IPDPS.2009.5161031.

[39] P Colella, D. Graves, T. Ligocki, D. Martin, D Modiano, D. Serafini, and B Van Straalen. *Chombo Software Package for AMR Applications-Design Document*. Tech. rep. Lawrence Berkeley National Laboratory, 2000.

[40] F. Commoner, A. Holt, S. Even, and A. Pnueli. "Marked directed graphs". In: *Journal of Computer and System Sciences* 5.5 (1971), pp. 511 –523. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(71)80013-2. URL: http://www.sciencedirect.com/science/article/pii/S0022000071800132.

[41] R. Courtland. "Intel strikes back [News]". In: *Spectrum, IEEE* 50.8 (2013), pp. –. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2013.6565547.

[42] L. Dagum and R. Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *Computing in Science and Engineering* 5 (1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313.

[43] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic ParallelizationScheduling and Automatic Parallelization*. 1st ed. Birkhäuser Boston, Mar. 2000. ISBN: 0817641491.

[44] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors". In: *SIAM Rev.* 51.1 (Feb. 2009), pp. 129–159. ISSN: 0036-1445. DOI: 10.1137/070693199.

[45] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *Proceedings of the 2008 ACM/IEEE*

conference on Supercomputing. SC '08. Austin, Texas: IEEE Press, 2008, 4:1–4:12. ISBN: 978-1-4244-2835-9.

[46] R. P. Dick. "TGFF: task graphs for free". In: *CODES/CASHE '98*. Seattle, Washington, United States, 1998. ISBN: 0-8186-8442-9.

[47] A. Dominguez, S. Udayakumaran, and R. Barua. "Heap Data Allocation to Scratch-pad Memory in Embedded Systems". In: *J. Embedded Comput.* 1.4 (Dec. 2005), pp. 521–540. ISSN: 1740-4460. URL: http://dl.acm.org/citation.cfm?id=1233791.1233799.

[48] J. Eker and J. Janneck. *Caltrop—Language report*. Technical Memorandum. http://www.gigascale.org/caltrop. University of California at Berkeley California, USA: Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, 2002.

[49] P. Eles, A. Doboli, P. Pop, and Z. Peng. "Scheduling with bus access optimization for distributed embedded systems". In: *IEEE Transactions on VLSI* (2000).

[50] C. Erbas, S. Cerav-Erbas, and A. Pimentel. "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design". In: *Evolutionary Computation, IEEE Transactions on* 10.3 (2006), pp. 358 –374. ISSN: 1089-778X. DOI: 10.1109/TEVC.2005.860766.

[51] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. "Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems". In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 29 (6 2010), pp. 911–924. ISSN: 0278-0070. DOI: http://dx.doi.org/10.1109/TCAD.2010.2048354. URL: http://dx.doi.org/10.1109/TCAD.2010.2048354.

[52] M. H. Foroozannejad, M. Hashemi, T. L. Hodges, and S. Ghiasi. "Look into details: the benefits of fine-grain streaming buffer analysis". In: *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*. LCTES '10. Stockholm, Sweden: ACM, 2010, pp. 27–36. ISBN: 978-1-60558-953-4. DOI: http://doi.acm.org/10.1145/1755888.1755894. URL: http://doi.acm.org/10.1145/1755888.1755894.

[53] M. Frigo and V. Strumpen. “Cache oblivious stencil computations”. In: *Proceedings of the 19th annual international conference on Supercomputing*. ICS ’05. Cambridge, Massachusetts: ACM, 2005, pp. 361–366. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088197.

[54] G. Gao, R. Govindarajan, and P. Panangaden. “Well-behaved dataflow programs for DSP computation”. In: *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*. Vol. 5. 1992, 561–564 vol.5. DOI: 10.1109/ICASSP.1992.226558.

[55] M. Gardner. “Mathematical games: The fantastic combinations of John Conway’s new solitaire game “life””. In: *Scientific American* 223.4 (1970), pp. 120–123.

[56] *Gecode: Generic constraint development environment*. URL: http://www.gecode.org.

[57] M. Geilen, T. Basten, and S. Stuijk. “Minimising buffer requirements of synchronous dataflow graphs with model checking”. In: *Design Automation Conference, 2005. Proceedings. 42nd*. 2005, pp. 819 –824. DOI: 10.1109/DAC.2005.193928.

[58] M. Geilen and T. Basten. “Requirements on the Execution of Kahn Process Networks”. English. In: *Programming Languages and Systems*. Ed. by P. Degano. Vol. 2618. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 319–334. ISBN: 978-3-540-00886-6. DOI: 10.1007/3-540-36575-3_22. URL: http://dx.doi.org/10.1007/3-540-36575-3_22.

[59] M. Geilen, T. Basten, and S. Stuijk. “Minimising buffer requirements of synchronous dataflow graphs with model checking”. In: *Proceedings of the 42nd annual Design Automation Conference*. DAC ’05. Anaheim, California, USA: ACM, 2005, pp. 819–824. ISBN: 1-59593-058-2. DOI: http://doi.acm.org/10.1145/1065579.1065796. URL: http://doi.acm.org/10.1145/1065579.1065796.

[60] P. Gelsinger. “Microprocessors for the new millennium: Challenges, opportunities, and new frontiers”. In: *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE*

International. 2001, pp. 22–25. DOI: 10.1109/ISSCC.2001.912412.

[61] S. V. Gheorghita and F. Catthoor. "System-scenario-based design of dynamic embedded systems". In: *ACM Trans. Des. Autom. Electron. Syst.* 14.1 (2009).

[62] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. D. Bosschere. "System-scenario-based design of dynamic embedded systems". In: *ACM Trans. Des. Autom. Electron. Syst.* 14 (1 2009), 3:1–3:45. ISSN: 1084-4309. DOI: http://doi.acm.org/10.1145/1455229.1455232. URL: http://doi.acm.org/10.1145/1455229.1455232.

[63] P. Ghysels, P. Kłosiewicz, and W. Vanroose. "Improving the arithmetic intensity of multigrid with the help of polynomial smoothers". In: *Numerical Linear Algebra with Applications* 19.2 (2012), pp. 253–267. ISSN: 1099-1506. DOI: 10.1002/nla.1808.

[64] J. Gladigau, A. Gerstlauer, C. Haubelt, M. StreubuÌˆ andhr, and J. Teich. "A system-level synthesis approach from formal application models to generic bus-based MPSoCs". In: *Embedded Computer Systems (SAMOS), 2010 International Conference on*. 2010, pp. 118 –125. DOI: 10.1109/ICSAMOS.2010.5642076.

[65] R. Govindarajan, G. R. Gao, and P. Desai. "Minimizing Buffer Requirements under Rate-Optimal Schedule in Regular Dataflow Networks". In: *J. VLSI Signal Process. Syst.* 31 (3 2002), pp. 207–229. ISSN: 0922-5773. DOI: 10.1023/A:1015452903532. URL: http://portal.acm.org/citation.cfm?id=598549.598651.

[66] A. Heinecke and M. Bader. "Parallel Matrix Multiplication Based on Space-filling Curves on Shared Memory Multicore Platforms". In: *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?* MAW '08. Ischia, Italy: ACM, 2008, pp. 385–392. ISBN: 978-1-60558-091-3. DOI: 10.1145/1366219.1366223. URL: http://doi.acm.org/10.1145/1366219.1366223.

[67] W. Heirman, D. Stroobandt, N. Miniskar, R. Wuyts, and F. Catthoor. "PinComm: Characterizing Intra-application Communication for the Many-Core Era". In: *Parallel and Distributed*

Systems (ICPADS), 2010 IEEE 16th International Conference on. 2010, pp. 500 –507. DOI: 10.1109/ICPADS.2010.56.

[68] W. Heirman, D. Stroobandt, N. Miniskar, R. Wuyts, and F. Catthoor. "PinComm: Characterizing Intra-application Communication for the Many-Core Era". In: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. 2010, pp. 500 –507. DOI: 10.1109/ICPADS.2010.56.

[69] T. C. Hu. "Parallel Sequencing and Assembly Line Problems". English. In: *Operations Research* 9.6 (1961), pp. 841–848. ISSN: 0030364X. URL: http://www.jstor.org/stable/167050.

[70] Intel Corporation. *Threading Building Blocks Reference Manual [online]*. URL: http://threadingbuildingblocks.org/documentation.

[71] J. R. Jackson. *Scheduling a production line to minimize maximum tardiness*. Tech. rep. DTIC Document, 1955.

[72] A. Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation (Systems on Silicon)*. 1st ed. Morgan Kaufmann, June 2003. ISBN: 1558609253.

[73] T. Johnson, T. A. Davis, and S. M. Hadfield. "A concurrent dynamic task graph". In: *Parallel Computing* 22.2 (1996), pp. 327 –333. ISSN: 0167-8191. DOI: 10.1016/0167-8191(95)00061-5.

[74] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. "An auto-tuning framework for parallel multicore stencil computations". In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. 2010, pp. 1 –12. DOI: 10.1109/IPDPS.2010.5470421.

[75] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. "Dynamic management of scratch-pad memory space". In: *Design Automation Conference, 2001. Proceedings*. 2001, pp. 690–695. DOI: 10.1109/DAC.2001.156226.

[76] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0.

[77] W. Kim and M. Voss. “Multicore Desktop Programming with Intel Threading Building Blocks”. In: *Software, IEEE* 28.1 (2011), pp. 23 –31. ISSN: 0740-7459. DOI: 10.1109/MS.2011.12.

[78] P. Laborie. “IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems”. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by W.-J. van Hoeve and J. Hooker. Vol. 5547. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 148–162. URL: http://dx.doi.org/10.1007/978-3-642-01929.

[79] K. Lagerstrom. “Design and implementation of an MP3 decoder - M.Sc Thesis”. PhD thesis. Chalmers University of Technology, Sweden, 2001.

[80] J. Laudon and D. Lenoski. “The SGI Origin: a ccNUMA highly scalable server”. In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pp. 241–251. ISSN: 0163-5964. DOI: 10.1145/384286.264206. URL: http://doi.acm.org/10.1145/384286.264206.

[81] J. Laurent, N. Julien, E. Senn, and E. Martin. “Functional Level Power Analysis: An Efficient Approach for Modeling the Power Consumption of Complex Processors”. In: *Proceedings of the conference on Design, automation and test in Europe - Volume 1*. DATE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 10666–. ISBN: 0-7695-2085-5. URL: http://portal.acm.org/citation.cfm?id=968878.968987.

[82] J. Laurent, N. Julien, E. Senn, and E. Martin. “Functional Level Power Analysis: An Efficient Approach for Modeling the Power Consumption of Complex Processors”. In: *Design, Automation and Test in Europe Conference and Exhibition* 1 (2004), p. 10666. ISSN: 1530-1591. DOI: http://doi.ieeecomputersociety.org/10.1109/DATE.2004.1268921.

[83] E. A. Lee and D. G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235 –1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[84] E. A. Lee. “The Problem with Threads”. In: *Computer* 39 (2006), pp. 33–42. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2006.180.

[85] E. A. Lee and D. G. Messerschmitt. “Static scheduling of synchronous data flow programs for digital signal processing”. In: *IEEE Trans. Comput.* 36 (1 1987), pp. 24–35. ISSN: 0018-9340. DOI: http://dx.doi.org/10.1109/TC.1987.5009446. URL: http://dx.doi.org/10.1109/TC.1987.5009446.

[86] C. L. Liu and J. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: http://doi.acm.org/10.1145/321738.321743.

[87] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. “The Jigsaw continuous sensing engine for mobile phone applications”. In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. SenSys ’10. Zürich, Switzerland: ACM, 2010, pp. 71–84. ISBN: 978-1-4503-0344-6. DOI: 10.1145/1869983.1869992.

[88] D. Ludovici, A. Strano, G. N. Gaydadjiev, L. Benini, and D. Bertozzi. “Design space exploration of a mesochronous link for cost-effective and flexible GALS NOCs”. In: *DATE*. 2010.

[89] M. Lukasiewycz, M. Streubühr, M. Glaß, C. Haubelt, and J. Teich. “Combined system synthesis and communication architecture exploration for MPSoCs”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’09. Nice, France: European Design and Automation Association, 2009, pp. 472–477. ISBN: 978-3-9810801-5-5. URL: http://dl.acm.org/citation.cfm?id=1874620.1874737.

[90] J. Luo and N. Jha. “Power-Efficient Scheduling for Heterogeneous Distributed Real-Time Embedded Systems”. In: *CAD of Integrated Circuits and Systems, IEEE* (2007).

[91] N. Matthys, C. Huygens, D. Hughes, S. Michiels, and W. Joosen. “A component and policy-based approach for efficient sensor network reconfiguration”. In: *Proceedings of the 13th IEEE International Symposium on Policies for Distributed Systems and Networks*. IEEE, 2012, pp. 53–60. URL: https://lirias.kuleuven.be/handle/123456789/350799.

[92] S. A. McKee. “Reflections on the Memory Wall”. In: *Proceedings of the 1st Conference on Computing Frontiers*. CF ’04. Ischia, Italy: ACM, 2004, pp. 162–. ISBN: 1-58113-741-9. DOI: 10.1145/977091.977115. URL: http://doi.acm.org/10.1145/977091.977115.

[93] J.-Y. Mignolet and R. Wuyts. “Embedded Multiprocessor Systems-on-Chip Programming”. In: *Software, IEEE* 26.3 (2009), pp. 34 –41. ISSN: 0740-7459. DOI: 10.1109/MS.2009.64.

[94] G. Moore. “Cramming More Components Onto Integrated Circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[95] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. “Buffer Sizing for Rate-Optimal Single-Rate Data-Flow Scheduling Revisited”. In: *Computers, IEEE Transactions on* 59.2 (2010), pp. 188 –201. ISSN: 0018-9340. DOI: 10.1109/TC.2009.155.

[96] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. “Buffer Sizing for Rate-Optimal Single-Rate Data-Flow Scheduling Revisited”. In: *Computers, IEEE Transactions on* 59.2 (2010), pp. 188 –201. ISSN: 0018-9340. DOI: 10.1109/TC.2009.155.

[97] P. Murthy and S. Bhattacharyya. “Shared buffer implementations of signal processing systems using lifetime analysis techniques”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 20.2 (2001), pp. 177 –198. ISSN: 0278-0070. DOI: 10.1109/43.908427.

[98] N. Z. Naqvi, D. Preuveneers, W. Meert, and Y. Berbers. “The Right Thing to Do: Automating Support for Assisted Living with Dynamic Decision Networks”. In: *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*. 2013, pp. 262–269. DOI: 10.1109/UIC-ATC.2013.65.

[99] R. H. B. Netzer and B. P. Miller. “What are race conditions?: Some issues and formalizations”. In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623. URL: http://doi.acm.org/10.1145/130616.130623.

[100] J. Parkhurst, J. Darringer, and B. Grundmann. "From Single Core to Multi-core: Preparing for a New Exponential". In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '06. San Jose, California: ACM, 2006, pp. 67–72. ISBN: 1-59593-389-1. DOI: 10.1145/1233501.1233516. URL: http://doi.acm.org/10.1145/1233501.1233516.

[101] J. Perez, R. Badia, and J. Labarta. "A dependency-aware task-based programming environment for multi-core architectures". In: *Cluster Computing, 2008 IEEE International Conference on*. 2008, pp. 142–151. DOI: 10.1109/CLUSTR.2008.4663765.

[102] J. Pino and E. Lee. "Hierarchical static scheduling of dataflow graphs onto multiple processors". In: *Acoustics, Speech, and Signal Processing, IEEE International Conference on* 4 (1995), pp. 2643–2646. DOI: http://doi.ieeecomputersociety.org/10.1109/ICASSP.1995.480104.

[103] F. J. Pollack. "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address)(Abstract Only)". In: *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 32. Haifa, Israel: IEEE Computer Society, 1999, pp. 2–. ISBN: 0-7695-0437-X. URL: http://dl.acm.org/citation.cfm?id=320080.320082.

[104] V. K. Prasanna. "Power-Aware Resource Allocation for Independent Tasks in Heterogeneous Real-Time Systems". In: *ICPADS*. 2002.

[105] A. Radulescu and A. van Gemund. "Fast and effective task scheduling in heterogeneous systems". In: *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*. 2000, pp. 229–238. DOI: 10.1109/HCW.2000.843747.

[106] A. Ramakrishnan, S. N. Z. Naqvi, Z. W. Bhatti, D. Preuveneers, and Y. Berbers. "Learning deployment trade-offs for self-optimization of Internet of Things applications". In: *Proceedings of the 10th International Conference on Autonomic Computing, ICAC 2013*. ACM, 2013, pp. 213–224. URL: https://lirias.kuleuven.be/handle/123456789/411273.

[107] K. Ramamritham, G. Fohler, and J. M. Adan. “Issues in the static allocation and scheduling of complex periodic tasks”. In: IEEE Computer Society, 1993.

[108] J. Reinders. *Intel threading building blocks*. First. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2007. ISBN: 9780596514808.

[109] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes. “Using Constraint Programming to Manage Configurations in Self-Adaptive Systems”. In: *Computer* 45.10 (2012), pp. 56–63. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2012.286.

[110] R. Schreiber and D. C. Cronquist. “Near-optimal allocation of local memory arrays”. In: *HP tenchnical report*. 2004.

[111] J. Shalf, S. Dosanjh, and J. Morrison. “Exascale Computing Technology Challenges”. In: *High Performance Computing for Computational Science – VECPAR 2010*. Ed. by J. Palma, M. Daydé, O. Marques, and J. Lopes. Vol. 6449. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 1–25. ISBN: 978-3-642-19327-9. DOI: 10.1007/978-3-642-19328-6_1. URL: http://dx.doi.org/10.1007/978-3-642-19328-6_1.

[112] J. Shirako, K. Sharma, and V. Sarkar. “Unifying Barrier and Point-to-Point Synchronization in OpenMP with Phasers”. In: *OpenMP in the Petascale Era*. Ed. by B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Müller. Vol. 6665. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 122–137. ISBN: 978-3-642-21486-8. DOI: 10.1007/978-3-642-21487-5.

[113] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. “Phasers: a unified deadlock-free construct for collective and point-to-point synchronization”. In: *Proceedings of the 22nd annual international conference on Supercomputing*. ICS ’08. Island of Kos, Greece: ACM, 2008, pp. 277–288. ISBN: 978-1-60558-158-3. DOI: 10.1145/1375527.1375568.

[114] A. Sinha and A. Chandrakasan. “JouleTrack-a Web based tool for software energy profiling”. In: *Design Automation Conference, 2001. Proceedings*. 2001, pp. 220 –225. DOI: 10.1109/DAC.2001.156139.

[115] J. Sjödin and C. von Platen. "Storage Allocation for Embedded Processors". In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '01. Atlanta, Georgia, USA: ACM, 2001, pp. 15–23. ISBN: 1-58113-399-5. DOI: 10.1145/502217.502221. URL: http://doi.acm.org/10.1145/502217.502221.

[116] J. Skeppstedt and P. Stenström. "Using Dataflow Analysis Techniques to Reduce Ownership Overhead in Cache Coherence Protocols". In: *ACM Trans. Program. Lang. Syst.* 18.6 (Nov. 1996), pp. 659–682. ISSN: 0164-0925. DOI: 10.1145/236114.236116. URL: http://doi.acm.org/10.1145/236114.236116.

[117] P. van Stralen and A. Pimentel. "Scenario-based design space exploration of MPSoCs". In: *Computer Design (ICCD), 2010 IEEE International Conference on*. 2010, pp. 305 –312. DOI: 10.1109/ICCD.2010.5647727.

[118] R. Strzodka, M. Shaheen, D. Pajak, and H. P Seidel. "Cache Accurate Time Skewing in Iterative Stencil Computations". In: *Parallel Processing (ICPP), 2011 International Conference on*. 2011, pp. 571–581. DOI: 10.1109/ICPP.2011.47.

[119] S. Stuijk, M. Geilen, and T. Basten. "Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs". In: *Computers, IEEE Transactions on* 57.10 (2008), pp. 1331 –1345. ISSN: 0018-9340. DOI: 10.1109/TC.2008.58.

[120] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications". In: *Embedded Computer Systems (SAMOS), 2011 International Conference on*. 2011, pp. 404 –411. DOI: 10.1109/SAMOS.2011.6045491.

[121] S. Stuijk, M. Geilen, and T. Basten. "Exploring Trade-offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs". In: *Proceedings of the 43rd Annual Design Automation Conference*. DAC '06. San Francisco, CA, USA: ACM, 2006, pp. 899–904. ISBN: 1-59593-381-6. DOI: 10.1145/1146909.1147138. URL: http://doi.acm.org/10.1145/1146909.1147138.

[122] V. Suhendra, C. Raghavan, and T. Mitra. “Integrated scratchpad memory optimization and task scheduling for MPSoC architectures”. In: CASES ’06.

[123] H. Sun, V. D. Florio, N. Gui, and C. Blondia. “Promises and Challenges of Ambient Assisted Living Systems”. In: *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*. ITNG ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1201–1207. ISBN: 978-0-7695-3596-8. DOI: 10.1109/ITNG.2009.169. URL: http://dx.doi.org/10.1109/ITNG.2009.169.

[124] H. Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb’s Journal* 30.3 (2005), pp. 202–210.

[125] H. Sutter and J. Larus. “Software and the Concurrency Revolution”. In: *Queue* 3.7 (Sept. 2005), pp. 54–62. ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. URL: http://doi.acm.org/10.1145/1095408.1095421.

[126] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. “The pochoir stencil compiler”. In: *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*. SPAA ’11. San Jose, California, USA: ACM, 2011, pp. 117–128. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989508.

[127] W. Thies and S. Amarasinghe. “An empirical characterization of stream programs and its implications for language and compiler design”. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. PACT ’10. Vienna, Austria: ACM, 2010, pp. 365–376. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854319. URL: http://doi.acm.org/10.1145/1854273.1854319.

[128] W. Thies, M. Karczmarek, and S. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Compiler Construction*. Ed. by R. Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 49–84.

[129] J. Torrellas, M. Lam, and J. L. Hennessy. "False sharing and spatial locality in multiprocessor caches". In: *Computers, IEEE Transactions on* 43.6 (1994), pp. 651–663. ISSN: 0018-9340. DOI: 10.1109/12.286299.

[130] S. Udayakumaran and R. Barua. "An Integrated Scratch-pad Allocator for Affine and Non-affine Code". In: *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*. DATE '06. Munich, Germany: European Design and Automation Association, 2006, pp. 925–930. ISBN: 3-9810801-0-6. URL: http://dl.acm.org/citation.cfm?id=1131481.1131740.

[131] D. Unat, X. Cai, and S. B. Baden. "Mint: realizing CUDA performance in 3D stencil methods with annotated C". In: *Proceedings of the international conference on Supercomputing*. ICS '11. Tucson, Arizona, USA: ACM, 2011, pp. 214–224. ISBN: 978-1-4503-0102-2. DOI: 10.1145/1995896.1995932.

[132] S. Verdoolaege, H. Nikolov, and T. Stefanov. "pn: a tool for improved derivation of process networks". In: *EURASIP J. Embedded Syst.* 2007.1 (Jan. 2007), pp. 19–19. ISSN: 1687-3955. DOI: 10.1155/2007/75947. URL: http://dx.doi.org/10.1155/2007/75947.

[133] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. "A co-design approach for embedded system modeling and code generation with UML and MARTE". In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.* 2009, pp. 226–231. DOI: 10.1109/DATE.2009.5090662.

[134] M. A. Viredaz and D. A. Wallach. "Power evaluation of a handheld computer". In: *Micro, IEEE* 23.1 (2003), pp. 66 –74. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1179900.

[135] M. Voss. *The Intel Threading Building Blocks flow graph is now fully supported [online]*. URL: http://software.intel.com/en-us/blogs/2011/09/08/the-intel-threading-building-blocks-flow-graph-is-now-fully-supported.

[136] P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. "Cyclo-dynamic dataflow". In: *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*. 1996, pp. 319–326. DOI: 10.1109/EMPDP.1996.500603.

[137] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. "Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization". In: *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*. Vol. 1. 2009, pp. 579 –586. DOI: 10.1109/COMPSAC.2009.82.

[138] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. "Efficient Computation of Buffer Capacities for Multi-rate Real-time Systems with Back-pressure". In: *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '06. Seoul, Korea: ACM, 2006, pp. 10–15. ISBN: 1-59593-370-0. DOI: 10.1145/1176254.1176260. URL: http://doi.acm.org/10.1145/1176254.1176260.

[139] M. Wiggers, M. Bekooij, and G. Smit. "Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication". In: 2008, pp. 183 –194. DOI: 10.1109/RTAS.2008.10.

[140] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[141] M. E. Wolf and M. S. Lam. "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991, pp. 30–44. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113449. URL: http://doi.acm.org/10.1145/113445.113449.

[142] C. Wong, F. Thoen, F. Catthoor, and D. Verkest. "System Design Automation". In: VDI-Verlag, 2001. Chap. Static Task Scheduling for Real-Time Embedded Systems, pp. 35–44.

[143] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. "Managing Dynamic Concurrent Tasks in Embedded Real-time Multimedia Systems". In: *Proceedings of the 15th International Symposium on System Synthesis*. ISSS '02. Kyoto, Japan: ACM, 2002, pp. 112–119. ISBN: 1-58113-576-9. DOI: 10.1145/581199.581226. URL: http://doi.acm.org/10.1145/581199.581226.

[144] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. "Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs". In: *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*. 2009, pp. 96 –105. DOI: 10.1109/ESTMED.2009.5336821.

[145] A. YarKhan, J. Kurzak, and J. Dongarra. *QUARK Users' Guide: QUeueing And Runtime for Kernels*. Tech. rep. Innovative Computing Laboratory, University of Tennessee, 2011.

[146] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. "Productivity and performance using partitioned global address space languages". In: *Proceedings of the 2007 international workshop on Parallel symbolic computation*. PASCO '07. London, Ontario, Canada: ACM, 2007, pp. 24–32. ISBN: 978-1-59593-741-4. DOI: 10.1145/1278177.1278183. URL: http://doi.acm.org/10.1145/1278177.1278183.

[147] E. Zitzler, J. Teich, and S. S. Bhattacharyya. "Evolutionary algorithms for the synthesis of embedded software". In: *IEEE Trans. Very Large Scale Integr. Syst.* 8 (4 2000), pp. 452–456. ISSN: 1063-8210. DOI: 10.1109/92.863627. URL: http://portal.acm.org/citation.cfm?id=349683.358382.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
zubairwadood.bhatti@cs.kuleuven.be